ILOG CPLEX 8.1

User's Manual

December 2002

© Copyright 2001, 2002 by ILOG

This document and the software described in this document are the property of ILOG and are protected as ILOG trade secrets. They are furnished under a license or non-disclosure agreement, and may be used or copied only within the terms of such license or non-disclosure agreement.
No part of this work may be reproduced or disseminated in any form or by any means, without the prior written permission of ILOG S.A.

Printed in France

Table of Contents

List of Figures 15

List of Tables 17

List of Examples 19

Preface **Meet ILOG CPLEX** 21

What Is ILOG CPLEX? 22

What You Need to Know 23

In This Manual 24

Examples Online 25

Notation in This Manual 26

Related Documentation 27

For More Information 29

Customer Support 29

Web Site 29

Chapter 1 **ILOG Concert Technology for C++ Users** 31

The Design of the ILOG CPLEX Concert Technology Library 31

Licenses 32

Compiling and Linking 32

Creating an Application with the Concert Technology Library 33

Modeling an Optimization Problem with Concert Technology 33

Modeling Classes 33
Data Management Classes 36
Solving Concert Technology Models with IloCplex 37
Extracting a Model 38
Solving a Model 39
Choosing an Optimizer 39
Accessing Solution Information 41
Querying Solution Data 43
Accessing Basis Information 43
Performing Sensitivity Analysis 44
Analyzing Infeasible Problems 44
Solution Quality 45
Modifying a Model 45
Deleting and Removing Modeling Objects 46
Changing Variable Type 47
Handling Errors 47
Example: Dietary Optimization 49
Program Description 51
Solving the Model with IloCplex 53
Complete Program 54

Chapter 2 ILOG Concert Technology for Java Users 59

Modeling with Concert Technology 60
Modeling with IloModeler 61
The Active Model 64
Building the Model 64
Solving the Model 66
Accessing Solution Information 68
Choosing an Optimizer 69
Solving a Single LP or QP 69
Solving Subsequent LPs or QPs in a MIP 70
Controlling ILOG CPLEX Optimizers 70

Parameters 70
More Solution Information 72
Dual Solution Information 72
Basis Information 72
Infeasible Solution Information 73
Solution Quality 74
Advanced Modeling with IloMPModeler 74
Modeling by Column 75
Solving the Diet Model with the Java API 76
Creating the Optimizer 76
Example: Diet.java Source Code 78
Modifying the Model 82

Chapter 3 The ILOG CPLEX Callable Library 83

Architecture of the ILOG CPLEX Callable Library 83
Licenses 84
Compiling and Linking 85
Using the Callable Library in an Application 85
Initialize the ILOG CPLEX Environment 85
Instantiate the Problem Object 86
Put Data in the Problem Object 86
Optimize the Problem 86
Change the Problem Object 87
Destroy the Problem Object 87
Release the ILOG CPLEX Environment 87
ILOG CPLEX Programming Practices 87
Variable Names and Calling Conventions 88
Data Types 89
Ownership of Problem Data 89
Copying in MIP and QP 90
Problem Size and Memory Allocation Issues 90
Status and Return Values 91

Symbolic Constants 91
Parameter Routines 91
Null Arguments 92
Row and Column References 92
Character Strings 93
Checking Problem Data 93
Callbacks 94
Portability 95
FORTRAN Interface 96
C++ Interface 97
Managing Parameters from the Callable Library 98
Example: Dietary Optimization 99
Program Description 101
Complete Program 102

Chapter 4 Further Programming Considerations 113

Tips for Successful Application Development 113
Prototype the Model 113
Identify Routines to Use 114
Test Interactively 114
Assemble Data Efficiently 114
Test Data 115
Choose an Optimizer 115
Program with a View toward Maintenance and Modifications 116
Using the Interactive Optimizer for Debugging 118
Eliminating Common Programming Errors 119
Check Your Include Files 120
Clean House and Try Again 120
Read Your Messages 120
Check Return Values 120
Beware of Numbering Conventions 120
Make Local Variables Temporarily Global 120

Solve the Problem You Intended 121
Special Considerations for Fortran 121
Tell Us 121

Chapter 5 Solving Linear Programming Problems 123

Choosing an Optimizer for Your LP Problem 124
Automatic Selection of Optimizer 125
Dual Simplex Optimizer 125
Primal Simplex Optimizer 125
Network Optimizer 126
Barrier Optimizer 126
Sifting Optimizer 126
Concurrent Optimizer 127
Tuning LP Performance. 127
Preprocessing 127
Starting from an Advanced Basis 129
Simplex Parameters 130
Diagnosing Performance Problems 134
Lack of Memory 134
Numerical Difficulties 135
Diagnosing LP Infeasibility 139
The Effect of Preprocessing on Feasibility 140
Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities 140
Interpreting Solution Statistics 141
Finding a Set of Irreducibly Inconsistent Constraints 143
Example: Using a Starting Basis in an LP Problem 148
Example ilolpex6.cpp 148
Example lpex6.c 150
Solving LP Problems with the Barrier Optimizer 156
Identifying LPs for Barrier Optimization 157
Using the Barrier Optimizer 159
Special Options 159

Controlling Crossover 160
Interpreting the Barrier Log File 160
Understanding Solution Quality from the Barrier LP Optimizer 163
Tuning Barrier Optimizer Performance 165
Overcoming Numerical Difficulties 169
Diagnosing Barrier Optimizer Infeasibility 172

Chapter 6 Solving Network-Flow Problems 175

Choosing an Optimizer: Network Considerations 175
Formulating a Network Problem 176
Example: Network Optimizer in the Interactive Optimizer 177
Understanding the Network Log File 178
Tuning Performance of the Network Optimizer 179
Solving Problems with the Network Optimizer 179
Network Extraction 180
Preprocessing and the Network Optimizer 181
Example: Using the Network Optimizer with the Callable Library 181
Complete Program: netex1.c 182
Solving Network-Flow Problems as LP Problems 188
Example: Network to LP Transformation 189
Complete Program: netex2.c 190

Chapter 7 Solving Quadratic Programming Problems 195

Identifying Convex Quadratic Programming Problems 195
Entering QPs 196
Saving QP Problems 198
Changing Problem Type in QPs 198
Changing Quadratic Terms 199
Optimizing QPs 201
Diagnosing QP Infeasibility 202
Example: Creating a QP, Optimizing, Finding a Solution 203
Example: iloqpex1.cpp 203

Example: QPex1.java 205
Example: Reading a QP from a File 215
Complete Program: qpex2.c 215

Chapter 8 Solving Mixed Integer Programming Problems 223

Sample: Stating a MIP Problem 224
Considering Preliminary Issues 225
Entering MIP Problems 225
Displaying MIP Problems 226
Changing Problem Type in MIPs 226
Using the Mixed Integer Optimizer 229
Feasibility and Optimality 229
Performance Features of the Mixed Integer Optimizer 231
Cuts 235
Parameters Affecting Cuts 237
Heuristics 238
Probing 238
Preprocessing: Presolver and Aggregator 239
Starting from a Solution 240
Priority Orders 240
Termination 241
Post-Solution Information in a MIP 243
Using Sensitivity Information in a MIP 243
Using Special Ordered Sets 244
Example: SOS Type 1 for Sizing a Warehouse 244
Declaring SOS Members 245
Setting Branching Priority for an SOS 245
Assigning SOS Weights 246
Using Semi-Continuous Variables 246
Progress Reports: Interpreting the Node Log 246
Troubleshooting MIP Performance Problems 250
Too Much Time at Node 0 250

Trouble Finding More than One Feasible Solution 251
Large Number of Unhelpful Cuts 251
Lack of Movement in the Best Node 251
Time Wasted on Overly Tight Optimality Criteria 252
Running Out of Memory 253
Difficulty Solving Subproblems 256
Subproblem Optimization 257
Example: Optimizing a Basic MIP Problem 258
Complete Program: ilomipex1.cpp 259
Complete Program: mipex1.c 260
Example: Reading a MIP Problem from a File 269
Complete Program: ilomipex2.cpp 269
Complete Program: mipex2.c 271
Example: Using SOS and Priority 275
Complete Program: ilomipex3.cpp 275
Complete Program: mipex3.c 277

Chapter 9 More About Using ILOG CPLEX 289

Managing Input & Output 290
Understanding File Formats 290
Managing Log Files: the Log File Parameter 295
Handling Message Channels: the Output-Channel Parameter 296
Handling Message Channels: Callable Library Routines 297
Example: Using the Message Handler 298
Using Query Routines in the Callable Library 308
Using Surplus Arguments for Array Allocations 308
Example: Using Query Routines 309
Complete Program: lpex7.c 310
Goals in IloCplex 319
The Goal Stack 325
Memory Management and Goals 326
Cuts and Goals 327

Injecting Solutions 332
Controlling Goal-Controlled Search 333
Search Limits 340
Using Callbacks 341
Diagnostic Callbacks 341
Control Callbacks for IloCplex 359
A Comparison of Goals and Callbacks 369
Using Parallel Optimizers 371
Threads 371
Nondeterminism 373
Clock Settings and Time Measurement 373
Using Parallel Optimizers in the Interactive Optimizer 374
Using Parallel Optimizers in the ILOG CPLEX Component Libraries 374
Parallel MIP Optimizer 375
Parallel Barrier Optimizer 379
Parallel Simplex Optimizer 379
CPLEX Runtime Development and Licensing Procedures 381
Types of ILM Runtime Licenses 381
Routines Used for Licensing 381
Summary 384

Appendix A ILOG CPLEX Parameter Names. 385

Appendix B Interactive Optimizer Commands 393

Managing Parameters in the Interactive Optimizer. 400
Saving a Parameter Specification File 400

Index 403

List of Figures

Figure 1.1 A View of Concert Technology for C++ Users 32
Figure 2.1 A View of Concert Technology for Java Users 60
Figure 3.1 A View of the ILOG CPLEX Callable Library 84
Figure 6.1 A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources 178
Figure 7.1 Maximize a Concave Objective Function, Minimize a Convex Objective Function . . 196
Figure 9.1 ILOG CPLEX Message Handling Routines 298

List of Tables

Table 1.1 Concert Technology Modeling Objects 37
Table 1.2 Optimizer Options 40
Table 1.3 Algorithm Status and Information About the Model 42
Table 2.1 Choosing Lower and Upper Bound Values 63
Table 2.2 Solution Status 67
Table 2.3 Algorithm Types 70
Table 2.4 Constants in IloCplex.PrimalPricing 71
Table 2.5 Classes with Parameters Defined by Integers. 71
Table 3.1 Special data types in the ILOG CPLEX Callable Library 89
Table 3.2 Default values of ILOG CPLEX Growth Parameters 90
Table 3.3 Callable Library Routines for Parameters in the ILOG CPLEX Environment 92
Table 5.1 LPMethod Parameter Values for Optimizer Choice 124
Table 5.2 Symbolic Names for LP Solution Methods 125
Table 5.3 DPriInd Parameter Values for Dual Simplex Pricing Algorithm 131
Table 5.4 PPriInd Parameter Values for Primal Simplex Pricing Algorithm 131
Table 5.5 ScaIndParameter Values for Scaling Methods 132
Table 5.6 CraInd Parameter Values for the Dual Simplex Optimizer 133
Table 5.7 CraInd Parameter Values for the Primal Simplex Optimizer 133
Table 5.8 BarCrossAlg Parameter Values 160
Table 5.9 BarDisplay Parameter Values 161
Table 5.10 Infeasibilities and Norms in the Log File of a Barrier Optimization 163
Table 5.11 Barrier Solution Quality Display 164

Table 5.12 BarStartAlg Parameter Values for Starting-Point Heuristics 168
Table 5.13 BarAlg Parameter Values for Barrier Optimizer Algorithm 170
Table 7.1 RootAlg Parameter Values 201
Table 8.1 Interactive Optimizer Display Options for MIP Problems 226
Table 8.2 Problem Type Definitions 227
Table 8.3 Parameters for Controlling Branch & Cut Strategy 232
Table 8.4 NodeSel Parameter Values for Node Search Type 233
Table 8.5 VarSel Parameter Values for Branching Variable Choice 234
Table 8.6 BrDir Parameter Values for Branching Direction Choice 234
Table 8.7 Parameters for Controlling Cuts 237
Table 8.8 Parameters for Controlling MIP Preprocessing 239
Table 8.9 Parameters for Branching Priority Order 241
Table 8.10 Parameters to limit MIP optimization 242
Table 8.11 Values of the MIP Display Parameter 246
Table 8.12 Values for the Node File Storage Parameter 255
Table 8.13 Values of RootAlg and NodeAlg Parameters 258
Table 9.1 Options for the convert Utility and Corresponding File Extensions 295
Table 9.2 Options for the Output-Channel Command 296
Table 9.3 Channels Directing Output to Screen or to a File 297
Table 9.4 Callback Macros 344
Table 9.5 Status of Nonzero Callbacks for MIPs 351
Table 9.7 Thread-Limit Parameters 371
Table 9.8 Parallel Optimizer Methods and Routines 375
Table A.1 Parameter Name Equivalents 386

List of Examples

ilodiet.cpp 54
Diet.java 78
diet.c 102
Example of an infeasible.iis file 145
Example of infeasible.iis with many rows and columns 146
Time-limited search for an IIS 147
ilolpex6.cpp 149
lpex6.c 151
netex1.c 182
netex2.c 190
iloqpex1.cpp 203
QPex1.java 206
qpex1.c 207
qpex2.c 215
ilomipex1.cpp 259
mipex1.c 261
ilomipex2.cpp 269
mipex2.c 271
ilomipex3.cpp 275
mipex3.c 277
lpex5.c 299
lpex7.c 310

ilogoalex1.cpp 321
ilogoalex2.cpp 328
ilogoalex3.cpp 335
ilolpex4.cpp 347
lpex4.c 352
iloadmipex5.cpp 365

Meet ILOG CPLEX

ILOG CPLEX offers C, C++ and Java libraries that solve linear programming (LP) and related problems. Specifically, it solves linearly constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function. The variables in the model may be declared as continuous or further constrained to take only integer values.

ILOG CPLEX comes in three forms to meet a wide range of users' needs:

- The ILOG CPLEX **Interactive Optimizer** is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The program consists of the file `cplex.exe` on Windows platforms or `cplex` on UNIX platforms.
- **Concert Technology** is a set of libraries offering an API that includes modeling facilities to allow the programmer to embed ILOG CPLEX optimizers in C++ and Java applications. The library is provided in files `ilocplex.lib`, `concert.lib` and `cplex.jar` on Windows platforms and in `libilocplex.a`, `libconcert.a` and `cplex.jar` on UNIX platforms, and makes use of the Callable Library (described next).
- The ILOG CPLEX **Callable Library** is a C library that allows the programmer to embed ILOG CPLEX optimizers in applications written in C, Visual Basic, Fortran or any other language that can call C functions. The library is provided in files `cplex81.lib` and `cplex81.dll` on Windows platforms and in `libcplex.a` and `libcplex81.so` or `libcplex81.sl` on UNIX platforms.

In this manual, the phrase **ILOG CPLEX Component Libraries** is used when referring equally to any of these libraries. While all libraries are callable, the term ILOG CPLEX Callable Library as used here refers specifically to the C library.

This preface introduces ILOG CPLEX, version 8.1. It includes the following sections:

- What Is ILOG CPLEX?
- What You Need to Know
- In This Manual
- Notation in This Manual
- Related Documentation
- For More Information

What Is ILOG CPLEX?

ILOG CPLEX is a tool for solving, first of all, *linear* optimization problems. Such problems are conventionally written like this:

Minimize (or maximize)	$c_1x_1 + c_2x_2 + \ldots + c_nx_n$
subject to	$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \quad \sim \quad b_1$
	$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \quad \sim \quad b_2$
	$\ldots$
	$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \quad \sim \quad b_m$
with these bounds	$l_1 \leq x_1 \leq u_1, \ \ldots, \ l_n \leq x_n \leq u_n$

where the relation ~ may be greater than or equal to, less than or equal to, or simply equal to, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.

When a linear optimization problem is stated in that conventional form, we customarily refer to its coefficients and values by these terms:

objective function coefficients	$c_1, \ \ldots, \ c_n$
constraint coefficients	$a_{11}, \ \ldots, \ a_{mn}$
right-hand side	$b_1, \ \ldots, \ b_m$
upper bounds	$u_1, \ \ldots, \ u_n$
lower bounds	$l_1, \ \ldots, \ l_n$
variables or unknowns	$x_1, \ \ldots, \ x_n$

In the most basic linear optimization problem, the variables of the objective function are *continuous* in the mathematical sense, with no gaps between real values. To solve such linear programming problems, ILOG CPLEX implements optimizers based on the simplex algorithms (both primal and dual simplex) as well as primal-dual logarithmic barrier algorithms and a sifting algorithm. These alternatives are explained more fully in Chapter 5, *Solving Linear Programming Problems.*

ILOG CPLEX can also handle certain problems in which the objective function is not linear but *quadratic*. (The constraints in such a problem are still linear.) Such problems are known as *quadratic programs* or QPs. Chapter 7, *Solving Quadratic Programming Problems* covers those kinds of problems.

ILOG CPLEX is also a tool for solving linear programming problems in which some or all of the variables must assume *integer* values in the solution. Such problems are known as *mixed integer programs* or MIPs because they may combine continuous and discrete (for example, integer) variables in the objective function and constraints. MIPs with linear objectives are referred to as *mixed integer linear programs* or MILPs, and MIPs with quadratic objective terms are referred to as *mixed integer quadratic programs* or MIQPs.

Within the category of mixed integer programs, we distinguish two kinds of discrete integer variables: if the integer values of the discrete variables must be either 0 (zero) or 1 (one), then we refer to them as *binary*; if the integer values are not restricted in that way, we refer to them as general integer variables. This manual explains more about the mixed integer optimizer in Chapter 8, *Solving Mixed Integer Programming Problems.*

ILOG CPLEX also offers a Network Optimizer aimed at a special class of linear problem with network structures. ILOG CPLEX can optimize such problems as ordinary linear programs, but if ILOG CPLEX can extract all or part of the problem as a network, then it will apply its more efficient Network Optimizer to that part of your problem and use the partial solution it finds there to construct an advanced starting point to optimize the rest of the problem. Chapter 6, *Solving Network-Flow Problems* offers more detail about how the ILOG CPLEX Network Optimizer works.

What You Need to Know

In order to use ILOG CPLEX effectively, you need to be familiar with your operating system, whether UNIX or Windows.

In this manual, we assume you are familiar with the concepts of mathematical programming, particularly linear programming. In case those concepts are new to you, the bibliography on page 30 in this preface indicates references to help you there.

This manual also assumes you already know how to create and manage files. In addition, if you are building an application that uses the Component Libraries, this manual assumes that you know how to compile, link, and execute programs written in a high-level language. The

Callable Library is written in the C programming language, while Concert Technology is written in C++ and Java. This manual also assumes that you already know how to program in the appropriate language and that you will consult a programming guide when you have questions in that area.

In This Manual

Chapter 1, *ILOG Concert Technology for C++ Users* introduces the Concert Technology Library. It provides an overview of the design of the library, explains modeling techniques, and offers an example of programming with the Concert Technology Library. It also provides information on controlling parameters.

Chapter 2, *ILOG Concert Technology for Java Users* explores the full range of features the ILOG CPLEX Java API offers to solve mathematical programming problems. An overview of the architecture is given, then techniques for creating models are explained.

Chapter 3, *The ILOG CPLEX Callable Library* introduces the ILOG CPLEX Callable Library. It describes the architecture of the product, explains the relation between the Interactive Optimizer and the Callable Library, and offers an example of programming with the Callable Library. It also provides an overview about the parameters you control in ILOG CPLEX.

Chapter 4, *Further Programming Considerations* provides tips on developing applications with ILOG CPLEX, suggests ways to debug your applications built around ILOG CPLEX, and provides a checklist to help avoid common programming errors.

Chapter 5, *Solving Linear Programming Problems* goes deeper into aspects of linear programming with ILOG CPLEX. It explains how to tune performance, how to diagnose infeasibility in a model, and how to use the primal-dual logarithmic barrier algorithm implemented in the ILOG CPLEX Barrier Optimizer on large, sparse linear programming problems. It also offers an example showing you how to start optimizing from an advanced basis.

Chapter 6, *Solving Network-Flow Problems* describes how to use the ILOG CPLEX Network Optimizer on linear programming problems based on a network model.

Chapter 7, *Solving Quadratic Programming Problems* takes up programming problems in which the objective function may be quadratic. It, too, includes examples.

Chapter 8, *Solving Mixed Integer Programming Problems* shows you how to handle MIPs. It particularly emphasizes performance tuning and offers a series of examples.

Chapter 9, *More About Using ILOG CPLEX* includes several sections on working with important aspects of the ILOG Component Libraries:

- Managing Input & Output explains how to enter mathematical programs efficiently and how to generate meaningful output from your ILOG CPLEX applications. It also lists the

available file formats for entering data into ILOG CPLEX and writing bases and solutions from ILOG CPLEX.

- Using Query Routines in the Callable Library shows how to access information about the model you currently have in memory through query routines of the Callable Library.
- Goals in ILOG CPLEX shows how to use goals to control the MIP search.
- Using Callbacks shows how to use callbacks to control the MIP search. A Comparison of Goals and Callbacks compares the two different approaches.
- Using Parallel Optimizers explains how to exploit parallel optimizers in case your hardware supports parallel execution.

Appendix B, *Interactive Optimizer Commands* lists the commands available in the ILOG CPLEX Interactive Optimizer with cross-references to examples of their use in this manual. It also provides an overview about controlling parameters with the Interactive Optimizer.

Examples Online

For the examples that we explain in the manual, we'll also show you the complete code for the solution, so that you can see exactly how ILOG CPLEX fits into your own applications. In case you prefer to study code online, you'll also find the complete code for these examples in a subdirectory of the standard distribution of ILOG CPLEX.

The following table describes all the examples in this manual and indicates where to find them, both online and in the manual:

Example	Source File	In This Manual
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `IloCplex`	`ilodiet.cpp`	*Example: Dietary Optimization* on page 49
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with `IloCplex`	`Diet.java`	*Example: Diet.java Source Code* on page 78
dietary optimization: building a model by rows (constraints) or by columns (variables), solving with the Callable Library	`diet.c`	*Example: Dietary Optimization* on page 99
linear programming: starting from an advanced basis	`ilolpex6.cpp` `lpex6.c`	*Example ilolpex6.cpp* on page 148 *Example lpex6.c* on page 150
network optimization: using the Callable Library	`netex1.c`	*Complete Program: netex1.c* on page 182

Example	Source File	In This Manual
network optimization: relaxing a network flow to an LP	`netex2.c`	*Complete Program: netex2.c* on page 190

quadratic programming: maximizing a QP	`iloqpex1.cpp`
	`QPex1.java`
	`qpex1.c`
		Example: iloqpex1.cpp on page 203
		Complete Program: QPex1.java on page 206
		Complete Program: qpex1.c on page 207

quadratic programming: reading a QP from a formatted file	`qpex2.c`	*Complete Program: qpex2.c* on page 215
mixed integer programming: optimizing a basic MIP	`ilomipex1.cpp` `mipex1.c`	*Complete Program: ilomipex1.cpp* on page 259 *Complete Program: mipex1.c* on page 260
mixed integer programming: reading a MIP from a formatted file	`ilomipex2.cpp` `mipex2.c`	*Complete Program: ilomipex2.cpp* on page 269 *Complete Program: mipex2.c* on page 271
mixed integer programming: using special ordered sets (SOS) and priority orders	`ilomipex3.cpp` `mipex3.c`	*Complete Program: ilomipex3.cpp* on page 275 *Complete Program: mipex3.c* on page 277
input and output: using the message handler	`lpex5.c`	*Complete Program: lpex5.c* on page 299
using query routines	`lpex7.c`	*Complete Program: lpex7.c* on page 310

using callbacks	`ilolpex4.c`
	`lpex4.c`
	`iloadmipex5.cpp`
		Complete Program: ilolpex4.cpp on page 347
		Complete Program: lpex4.c on page 352
		Complete Program: iloadmipex5.cpp on page 365

Notation in This Manual

Like the reference manual, this manual uses the following conventions:

- Important ideas are *italicized* the first time they appear.
- The names of C routines and parameters in the ILOG CPLEX Callable Library begin with `CPX`; the names of C++ and Java classes in the ILOG CPLEX Concert Technology Library begin with `Ilo`; and both appear in `this typeface`, for example: `CPXcopyobjnames()` or `IloCplex`.
- Where specific language usage (C++, Java, C, and so on) is unimportant and the effect on the optimization algorithms is to be emphasized, the names of ILOG CPLEX parameters are given as their Concert Technology variant. Appendix A, *ILOG CPLEX Parameter Names*, contains a table showing the correspondence of these names to the

Callable Library and the Interactive Optimizer. Full information about CPLEX parameters is found in the *ILOG CPLEX Reference Manual, Appendix A*.

- Text that is entered at the keyboard or displayed on the screen and commands and their options available through the Interactive Optimizer appear in `this typeface`, for example, `set preprocessing aggregator n`.
- Values that you must fill in (for example, the value to set a parameter) also appear in the same typeface as the command but slanted to indicate you must supply an appropriate value; for example, `set simplex refactor i` indicates that you must fill in a value for *i*.
- Matrices are denoted in two ways:
  - In printed material where superscripts and bold type are available, we denote the product of A and its transpose like this: $\boldsymbol{AA^T}$. The superscript T indicates the matrix transpose.
  - In computer-generated samples, such as log files, where only ASCII characters are available, we denote the product of A and its transpose like this: `A*A'`. The asterisk (*) indicates matrix multiplication, and the prime (') indicates the matrix transpose.

Related Documentation

The complete documentation set for ILOG CPLEX 8.1 consists of the following materials:

- **ILOG CPLEX 8.1 Getting Started.** We recommend that new ILOG CPLEX users start with this manual. It introduces ILOG CPLEX through the interactive optimizer, and contains tutorials on ILOG CPLEX Concert Technology for C++ and Java applications and the ILOG CPLEX Callable Library.

  The *ILOG CPLEX 8.1 Getting Started* manual is supplied in HTML form, in Microsoft compiled HTML help (.CHM) form, and as a PDF file.

- **ILOG CPLEX 8.1 User's Manual.** This manual explains the topics covered in the Getting Started Manual in greater depth, with individual chapters on:
  - LP (Linear Programming) problems;
  - Network-Flow problems;
  - QP (Quadratic Programming) problems; and
  - MIP (Mixed Integer Programming) problems.

  There is also detailed information on:
  - managing input and output,
  - using query routines,

- using callbacks, and
- using parallel optimizers.

The *ILOG CPLEX 8.1 User's Manual* is supplied in HTML form, in Microsoft compiled HTML help (.CHM) form, and as a PDF file.

- **ILOG CPLEX 8.1 Reference Manual.** This manual supplies detailed definitions of the classes and functions in the ILOG CPLEX callable library and the Concert Technology for C++ ILOG CPLEX Library. It is available only in online form as HTML, Microsoft compiled HTML help (.CHM) form, and as a PDF file. You will find the searching facilities and cross-links of these formats very useful.
- **ILOG CPLEX 8.1 Java Reference Manual.** This manual supplies detailed definitions of the Concert Technology interfaces and ILOG CPLEX Java classes. It is available only in online form as HTML, and Microsoft compiled HTML help (.CHM) form.
- **ILOG License Manager (ILM).** ILOG products are protected by the ILOG License Manager. Before you can use ILOG CPLEX, you need to set up the ILM. This online documentation explains how to do it step-by-step, for different platforms. It is in HTML form, included with your distribution.
- **ILOG Concert Technology Documentation Kit.** This includes the *ILOG Concert Technology Reference Manual*, which documents the classes, methods, and functions of the Concert Technology library; the *ILOG Concert Technology User's Manual*, which provides examples that show how to use Concert Technology to model problems; the *ILOG Concert Technology Hybrid Optimizers User's Guide & Reference*, which documents the class `IloLinConstraint` and shows how to use ILOG's main algorithm classes, `IloSolver` and `IloCplex` in cooperation; and the *ILOG Concert Technology Migration Guide*, which shows how to translate applications created in previous versions of ILOG products to Concert Technology.

Online Documentation

The online information files are distributed with the ILOG CPLEX library files and can be found in `/doc` and its sub-directories.

For More Information

ILOG offers technical support and comprehensive Web sites for its products.

Customer Support

For technical support of ILOG CPLEX, you should contact your local distributor, or, if you are a direct ILOG customer, contact the nearest regional office:

Region	E-mail	Telephone	Fax
France	cplex-support@ilog.fr	0 800 09 27 91 (numéro vert) +33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Germany	cplex-support@ilog.de	+49 6172 40 60 33	+49 6172 40 60 10
Spain	cplex-support@ilog.es	+34 91 710 2480	+34 91 372 9976
United Kingdom	cplex-support@ilog.co.uk	+44 (0)1344 661600	+44 (0)1344 661601
Other European countries	cplex-support@ilog.fr	+33 (0)1 49 08 35 62	+33 (0)1 49 08 35 10
Japan	cplex-support@ilog.co.jp	+81 3 5211 5770	+81 3 5211 5771
Singapore	cplex-support@ilog.com.sg	+65 773 06 26	+65 773 04 39
USA	cplex-support@ilog.com	1-877-ILOG-TECH 1-877-456-4832 (toll free) or 1-650-567-8080	+1 650 567 8001

We encourage you to use e-mail for faster, better service.

Web Site

The ILOG CPLEX Web site at `http://www.ilog.com/products/cplex/` offers product descriptions, press releases, and contact information. It lists services, such as training, maintenance, technical support, and outlines special programs. In addition, it links you to an `ftp` site where you can pick up examples.

The technical support pages contain FAQ (Frequently Asked/Answered Questions) and the latest patches for some of our products. Access to these pages is restricted to owners of an ongoing maintenance contract. The maintenance contract number and the name of the person this contract is sent to in your company will be needed for access, as explained on the login page.

All three of the following sites contain the same information, but access is localized, so we recommend that you connect to the site corresponding to your location and select the "support" page from the home page.

- The Americas: `http://www.ilog.com`
- Asia & Pacific nations: `http://www.ilog.com.sg`
- Europe, Africa, and Middle East: `http://www.ilog.fr`

On those Web pages, you will find additional information about ILOG CPLEX in technical papers that have also appeared at industrial and academic conferences.

Further Reading

In case you want to know more about optimization and mathematical or linear programming, here is a brief selection of printed resources:

Williams, H. P. *Model Building in Mathematical Programming*, 4th ed. New York: John Wiley & Sons, 1999. This textbook includes many examples of how to design mathematical models, including linear programming formulations. (How you formulate your model is at least as important as what ILOG CPLEX does with it.) It also offers a description of the branch & bound algorithm.

Nemhauser, George L. and Laurence A. Wolsey, *Integer and Combinatorial Optimization*, New York: John Wiley & Sons, 1999. A reprint of the 1988 edition. A widely cited reference about integer programming, this book explains the branch & bound algorithm in detail.

Gill, Philip E., Walter Murray, and Margaret H. Wright, *Practical Optimization*. New York: Academic Press, 1982 reprint edition. This book covers, among other topics, quadratic programming.

1

ILOG Concert Technology for C++ Users

This chapter describes how to write C++ programs using the ILOG CPLEX Concert Technology library for C++ users. It includes sections on:

- The Design of the ILOG CPLEX Concert Technology Library, including information on licensing and on compiling and linking your programs
- Creating an Application with the Concert Technology Library
- Modeling an Optimization Problem with Concert Technology
- Solving Concert Technology Models with `IloCplex`
- Accessing Solution Information
- Modifying a Model
- Handling Errors
- Example: Dietary Optimization

The Design of the ILOG CPLEX Concert Technology Library

Figure 1.1, *A View of Concert Technology for C++ Users* shows a program using the ILOG CPLEX Concert Technology Library to solve optimization problems. The optimization part of the user's application program is captured in a set of interacting C++ or

objects that the application creates and controls. These objects can be divided into two categories:

1. **Modeling objects** are used to define the optimization problem. Generally an application creates several modeling objects to specify the optimization problems. Those objects are grouped into an `IloModel` object representing the complete optimization problem.
2. **`IloCplex` objects** are used to solve models created with the modeling objects. An `IloCplex` object reads a model and extracts its data to the appropriate representation for the ILOG CPLEX optimizer. Then the `IloCplex` object is ready to solve the model it extracted and be queried for solution information.

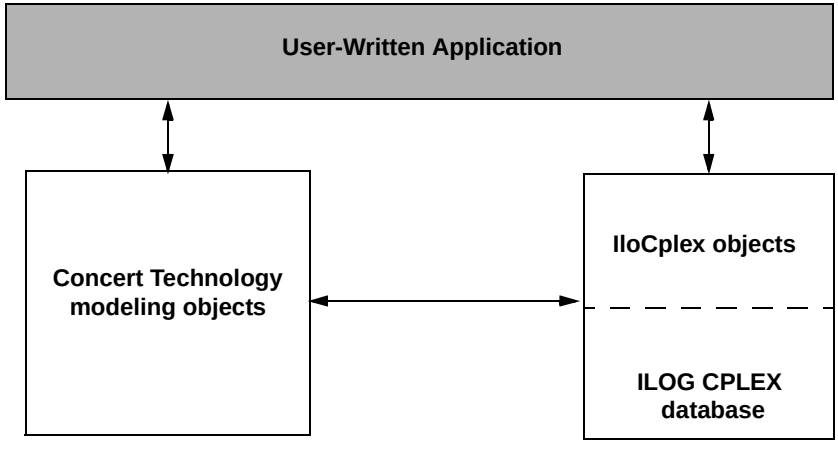

Figure 1.1 *A View of Concert Technology for C++ Users*

Licenses

ILOG CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run any application program that calls ILOG CPLEX, you must have established a valid license that it can read. Licensing instructions are provided to you separately when you buy or upgrade ILOG CPLEX. Contact your local ILOG support department if this information has not been communicated to you or if you find that you need help in establishing your ILOG CPLEX license.

Compiling and Linking

Compilation and linking instructions are provided with the files that come in the standard distribution of ILOG CPLEX for your computer platform. Check the `readme.html` file for details.

Creating an Application with the Concert Technology Library

The remainder of this chapter is organized by the steps most applications are likely to follow.

- First the model to be solved must be created. With Concert Technology this is done independently of ILOG CPLEX. A short introduction to model creation is provided in *Modeling an Optimization Problem with Concert Technology* on page 33. A more comprehensive discussion can be found in the *ILOG Concert Technology User's Manual*.
- When the model is ready to be solved, it is handed over to ILOG CPLEX for solving. The process for doing this is shown in *Solving Concert Technology Models with IloCplex* on page 37, which includes a survey of the `IloCplex` interface for controlling the optimization. Individual controls are discussed in the chapters explaining the individual optimizers.
- *Accessing Solution Information* on page 41, shows you how to access and interpret results from the optimization after solving the model.
- After analyzing the results, you may make changes to the model and study their effect. The way to perform such changes and how ILOG CPLEX deals with them is explained in *Modifying a Model* on page 45.
- *Handling Errors* on page 47, discusses the error handling and debugging support provided by Concert Technology and ILOG CPLEX.
- In *Example: Dietary Optimization* on page 49, an example program is presented.

Not covered in this chapter are advanced functions, such as the use of goals or callbacks for querying data about an ongoing optimization and for controlling the optimization itself. Goals, callbacks and other advanced functions are discussed in Chapter 9, *More About Using ILOG CPLEX*.

Modeling an Optimization Problem with Concert Technology

In this section we will only give a brief introduction to using Concert Technology for modeling optimization problems to be solved by `IloCplex`. For a more complete overview, see the *ILOG Concert Technology User's Manual*.

Modeling Classes

A Concert Technology model consists of a set of C++ objects. Each variable, each constraint, each SOS set, and the objective function in a model are represented by an object of the appropriate Concert Technology class. We refer to these objects as modeling objects.

Creating the Environment—The IloEnv Object

Before creating modeling objects, an object of class `IloEnv` must be constructed. We refer to this object as the environment object. It is constructed with the statement:

```
IloEnv env;
```

which is usually the first Concert Technology statement in an application. At the end, the environment must be closed by calling:

```
env.end();
```

This is usually the last Concert Technology statement in an application. The `end()` method must be called because, like most Concert Technology classes, the `IloEnv` class is a handle class. This means that `IloEnv` objects are really only pointers to implementation objects. Implementation objects are destroyed by calling the `end()` method. Failing to call the `end()` method can result in memory leaks. Please see the *ILOG Concert Technology User's Manual* and the *ILOG Concert Technology Reference Manual* for more details about handle classes in Concert Technology.

Users familiar with the ILOG CPLEX Callable Library are cautioned not to confuse the Concert Technology environment object with the ILOG CPLEX environment object of type `CPXENVptr`, used for setting ILOG CPLEX parameters. Such an object is not needed with Concert Technology, as parameters are handled directly by each instance of the `IloCplex` class. Thus, when talking about the environment in Concert Technology, we always refer to the object of class `IloEnv` required for all other Concert Technology objects.

Defining Variables and Expressions—The IloNumVar Object

Probably the first modeling class you will need is `IloNumVar`. Objects of this class represent modeling variables. They are described by the lower and upper bound for the variable, and a type which can be one of `ILOFLOAT`, `ILOINT`, or `ILOBOOL` for continuous, integer, or Boolean variables, respectively. The following constructor creates an integer variable with bounds -1 and 10:

```
IloNumVar myIntVar(env, -1, 10, ILOINT);
```

The `IloNumVar` class provides methods that allow querying of the data needed to specify a variable. However, only bounds can be modified. Concert Technology provides a modeling object class `IloConversion` to change the type of a variable. This allows you to use the same variable with different types in different models.

Variables are usually used to build up expressions, which in turn are used to define the objective or constraints of the optimization problem. An expression can be explicitly written, as in

```
1*x[1] + 2*x[2] + 3*x[3]
```

where `x` is assumed to be an array of variables (`IloNumVarArray`). Expressions can also be created piece by piece, with a loop:

```
IloExpr expr(env);
```

```
for (int i = 0; i < x.getSize(); ++i)
  expr += data[i] * x[i];
```

Whenever possible, build your expressions in terms of data that is either integer or double-precision (64 bit) floating point. Single-precision (32 bit) floating point data should be avoided as it can result in unnecessarily ill conditioned problems. For more information, refer to *Numerical Difficulties* on page 135.

While Concert Technology supports very general expressions, only linear, piecewise linear and quadratic expressions can be used in models to be solved with `IloCplex`. When you are done using an expression object (that is, you created a constraint with it) you need to delete it by calling its method `end()`, for example:

```
expr.end();
```

Declaring the Objective—The IloObjective Object

Objects of class `IloObjective` represent objective functions of optimization models. `IloCplex` may only handle models with at most one objective function, though the modeling API provided by Concert Technology does not impose this restriction. An objective function is specified by creating an instance of `IloObjective`. For example:

```
IloObjective obj(env,
                 1*x[1] + 2*x[2] + 3*x[3],
                 IloObjective::Minimize);
```

defines the objective to minimize the expression `1*x[1] + 2*x[2] + 3*x[3]`.

Adding Constraints—The IloRange Object

Similarly, objects of class `IloRange` represent constraints of the form `lower bound` ≤ `expression` ≤ `upper bound`. Any floating point value or `+/-IloInfinity` can be used for the bounds. For example:

```
IloRange r1(env, 3.0, x[1] + x[2], 3.0);
```

defines the constraint `x[1] + x[2] == 3.0`.

Formulating a Problem—The IloModel Object

To formulate a full optimization problem, the objects that are part of it need to be selected. This is done by adding them to an instance of `IloModel`, the class used to represent optimization problems. For instance:

```
IloModel model(env);
model.add(obj);
model.add(r1);
```

defines a model consisting of the objective `obj`, constraint `r1`, and all the variables they use. Notice that variables need not be added to a model explicitly, as they are implicitly considered if any of the other modeling objects in the model use them. However, variables may be explicitly added to a model if desired.

For convenience, Concert Technology provides the functions `IloMinimize` and `IloMaximize` to define minimization and maximization objective functions. Also, operators `<=, ==`, and `<=` are overloaded to create `IloRange` constraints. This allows us to rewrite the above examples in a more compact and readable way:

```
IloModel model(env);
model.add(IloMinimize(env, 1*x[1] + 2*x[2] + 3*x[3]);
model.add(x[1] + x[2] == 3.0);
```

With this notation the C++ variables `obj` and `r1` need not be created.

The `IloModel` class is itself a class of modeling objects. Thus, one model can be added to another. A possible use of this is to capture different scenarios in different models, all of which are extensions to a core model. The core model could be represented as an `IloModel` object added to the `IloModel` objects that represent the individual scenarios.

Data Management Classes

Usually the data describing an optimization problem must be collected before or during the creation of the Concert Technology representation of the model. Though in principle modeling does not depend on how the data is generated and represented, this task may be facilitated by using the array or set classes provided by Concert Technology.

For example, objects of class `IloNumArray` can be used to store numerical data in arrays. Elements of the class `IloNumArray` can be accessed like elements of standard C++ arrays, but the class also offers a wealth of additional functions. For example, Concert Technology arrays are extensible; in other words they transparently adapt to the required size when new elements are added using the method `add()`. Conversely, elements can be removed from anywhere in the array with the method `remove()`. Concert Technology arrays also provide debugging support when compiled in debug mode by using `assert` to ensure that no element beyond the array bounds is accessed. Input and output operators (that is, operator `<<` and operator `>>`) are provided for arrays. For example, the code:

```
IloNumArray data(env, 3, 1.0, 2.0, 3.0);
cout << data << endl;
```

produces the following output:

```
[1.0, 2.0, 3.0]
```

When you are done using an array and want to reclaim its memory, call method `end()`, for example, `data.end()`. However, when ending the environment, all memory of arrays belonging to the same environment is returned to the system as well. Thus, in practice you do not need to call `end()` on an array (or any other Concert Technology object) just before calling `env.end()`.

The constructor for arrays specifies that an array of size 3 with elements 1.0, 2.0, and 3.0 is constructed. This output format can be read back in with, for example:

```
cin >> data;
```

The example at the end of this chapter takes advantage of this function and reads the problem data from a file.

Finally, we want to point out that Concert Technology provides the template class `IloArray<X>` to create array classes for your own type X. This can be used to generate multidimensional arrays. All the functions described above are supported for `IloArray` classes except for input/output, which depends on the input and output operator being defined for type X.

Solving Concert Technology Models with IloCplex

ILOG CPLEX generally does not need to be involved while you create your model. However, once the model is set up, it is time to create your `cplex` object, that is, an instance of the class `IloCplex`, to be used to solve the model. `IloCplex` is a class derived from `IloAlgorithm`.. There are other Concert Technology algorithm classes, also derived from `IloAlgorithm`.. Some models might also be solved by using other algorithms, such as the class `IloSolver` for constraint programming, or by using a hybrid algorithm consisting of both `IloSolver` and ILOG CPLEX. Some models, on the other hand, cannot be solved with ILOG CPLEX.

The makeup of the model determines whether or not ILOG CPLEX can be used to solve it. More precisely, in order to be handled by `IloCplex` objects, a model may only consist of modeling objects of the following classes:

Table 1.1 *Concert Technology Modeling Objects*

To model:	Use:
numerical variables	objects of class `IloNumVar`, as long as they are not constructed with a list of feasible values
semi-continuous variable	objects of class `IloSemiContVar`
linear objective functions	objects of class `IloObjective` with linear or piecewise linear expressions
quadratic objective functions	objects of class `IloObjective` with quadratic expressions
linear constraints	objects of class `IloRange` with linear or piecewise linear expressions
variable type conversions	objects of class `IloConversion`
special ordered sets of type 1	objects of class `IloSOS1`
special ordered sets of type 2	objects of class `IloSOS2` and
constraints	objects of class `IloAnd.`

For a description of special ordered sets see *Using Special Ordered Sets* on page 244. The last class, `IloAnd`, is listed for completeness only and is generally not used with ILOG CPLEX, except with the class `IloSolution`, as described in the *ILOG Concert Technology User's Manual*.

Extracting a Model

In this manual we describe only one optimization model and use only one instance of `IloCplex` at a time to solve the model. Consequently, we talk about these as the model and the `cplex` object. It should be noted, however, that in Concert Technology an arbitrary number of models and algorithm objects can be created. The `cplex` object can be created using the constructor:

```
IloCplex cplex(env);
```

To use it to solve the model, the model must first be extracted to `cplex` by calling:

```
cplex.extract(model);
```

This method copies the data from the model into the appropriate optimized data structures, which ILOG CPLEX uses for solving the problem. It does so by extracting each of the modeling objects added to the model and each of the objects referenced by them. For every extracted modeling object, corresponding data structures are created internally in the `cplex` object. For readers familiar with the sparse matrix representation used internally by ILOG CPLEX, a variable becomes a column and a constraint becomes a row. As we will discuss later, these data structures are kept synchronized with the modeling objects even if the modeling objects are modified.

If you consider a variable to be part of your model, even though it is not (initially) used in any constraint, you should add this variable explicitly to the model. This ensures that the variable will be extracted. This may also be important if you query solution information for the variable, since solution information is available only for modeling objects that are known to ILOG CPLEX because they have been extracted from a model.

If you feel uncertain about whether or not an object will be extracted, you can add it to the model to be sure. Even if an object is added multiple times, it will only be extracted once and thus will not slow the solution process down.

Since the sequence of creating the `cplex` object and extracting the model to it is such a common one, `IloCplex` provides the shortcut:

```
IloCplex cplex(model);
```

This is completely equivalent to separate calls and ensures that the environment used for the `cplex` object will be the same as that used for the model when it is extracted, as required by Concert Technology. The shortcut uses the environment from the model to construct the `cplex` object before extraction.

Solving a Model

Once the model is extracted to the `cplex` object, you are ready to solve it. This is done by calling

```
cplex.solve();
```

For most problems this is all that is needed for solving the model. Nonetheless, ILOG CPLEX offers a variety of controls that allow you to tailor the solution process for your specific needs.

Choosing an Optimizer

Solving the extracted model with ILOG CPLEX involves solving one or a series of LPs or QPs:

- Only one LP or QP needs to be solved if the extracted model is continuous itself, that is, if it does not contain integer, Boolean, semi-continuous or semi-integer variables, SOS, or piecewise linear functions. Chapter 5, *Solving Linear Programming Problems* discusses the algorithms available for solving LPs. Similarly, Chapter 7, *Solving Quadratic Programming Problems*, discusses the algorithms available for solving QPs.
- In all other cases, the extracted problem that ILOG CPLEX sees is indeed a MIP and, in general, a series of LPs or QPs needs to be solved. Method `cplex.isMIP()` returns `IloTrue` in such a case. Chapter 8, *Solving Mixed Integer Programming Problems* discusses the algorithms applied.

The optimizer option used for solving the first LP or QP (whether it is the only one or just the first in a series of problems) is controlled by setting the root algorithm parameter:

```
cplex.setParam(IloCplex::RootAlg, alg);
```

where `alg` is a member of the nested enumeration type:

```
enum IloCplex::Algorithm {
  Primal,
  Dual,
  Barrier,
  Network,
  Sifting,
  Concurrent
};
```

Note that the last two choices, `Sifting` and `Concurrent`, are not available for QP models. As a nested enumeration type, the fully qualified names that must be used in the program are

IloCplex::Primal, IloCplex::Dual, and so on. Table 1.2 displays the meaning of the optimizer options defined by IloCplex::Algorithm.

Table 1.2 *Optimizer Options*

IloCplex::Primal	use the primal simplex algorithm
IloCplex::Dual	use the dual simplex algorithm
IloCplex::Barrier	use the barrier algorithm. The type of crossover performed after the barrier algorithm is determined by parameter IloCplex::BarCrossAlg.
IloCplex::Network	use the primal network simplex algorithm on an embedded network followed by the dual simplex algorithm for LPs and the primal simplex algorithm for QPs on the entire problem
IloCplex::Sifting	use the sifting algorithm
IloCplex::Concurrent	use multiple algorithms concurrently on a multiprocessor system

If the extracted model requires the solution of more than one LP or QP, the algorithm for solving all but the first is controlled by the NodeAlg parameter:

```
cplex.setParam(IloCplex::NodeAlg, alg).
```

Controlling ILOG CPLEX Optimizers

Though ILOG CPLEX defaults will prove sufficient to solve most of the problems, ILOG CPLEX offers a variety of parameters to control various algorithmic choices. ILOG CPLEX parameters can assume values of type bool, num, int, and string. IloCplex provides four categories of parameters that are listed in the nested enumeration types IloCplex::BoolParam, IloCplex::IntParam, IloCplex::NumParam, IloCplex::StringParam.

To access the *current* value of a parameter that interests you from the Concert Technology Library, use the method getParam. To access the default value of a parameter, use the method getDefault. Use the methods getMin and getMax to access the minimum and maximum values of num and int type parameters.

Some integer parameters are tied to nested enumerations that define symbolic constants for the values the parameter may assume.

In particular, these enumeration types are: IloCplex::MIPEmphasisType, , IloCplex::VariableSelect, IloCplex::NodeSelect, IloCplex::PrimalPricing, IloCplex::DualPricing., and IloCplex::BranchDirection. They are used for parameters IloCplex::MIPEmphasis, IloCplex::VarSel, IloCplex::NodeSel, IloCplex::PPriInd, IloCplex::DPriInd, and IloCplex::BrDir, respectively. Only the parameter IloCplex::MIPEmphasis may be of importance for general use.

There are, of course, routines in the Concert Technology Library to set these parameters. Use the following methods to set the values of ILOG CPLEX parameters:

```
IloCplex::setParam(BoolParam, value);
IloCplex::setParam(IntParam, value);
IloCplex::setParam(NumParam, value);
IloCplex::setParam(StringParam, value);
```

For example, the numerical parameter `IloCplex::EpOpt` controlling the optimality tolerance for the simplex algorithms can be set to 0.0001 by calling

```
cplex.setParam(IloCplex::EpOpt, 0.0001);
```

The *ILOG CPLEX Reference Manual* documents the type of each parameter (`bool`, `int`, `num`, `string`) along with the Concert Technology enumeration value, symbolic constant, and reference number representing the parameter.

The method `setDefaults` *resets all parameters* (except the log file) to their default values, including the ILOG CPLEX callback functions. This routine resets the callback functions to `NULL`.

When solving MIPs, additional controls of the solution process are provided. Priority orders and branching directions can be used to control the branching in a static way. These are discussed in *Heuristics* on page 238. These controls are static in the sense that they allow you to control the solution process based on data that does not change during the solution and can thus be set up before solving the model.

Dynamic control of the solution process of MIPs is provided through goals or control callbacks. They are discussed in *Goals in IloCplex* on page 319, and in *Using Callbacks* on page 341. Goals and callbacks allow you to control the solution process based on information that is generated during the solution process.

Accessing Solution Information

Information about:

- solution feasibility
- solution variables
- basis information
- solution quality

can be accessed with the methods described in this section.

Accessing Solution Status

Calling `cplex.solve()` returns a Boolean indicating whether or not a feasible solution (but not necessarily the optimal one) has been found. To obtain more of the information about the model that ILOG CPLEX found during the call to the `solve()` method, `cplex.getStatus()` can be called. It returns a member of the nested enumeration type:

```
enum IloAlgorithm::Status {
  Unknown,
  Feasible,
  Optimal,
  Infeasible,
  Unbounded,
  InfeasibleOrUnbounded,
  Error
};
```

Notice that the fully qualified names have the `IloAlgorithm` prefix. Table 1.3 shows what the possible return statuses mean for the extracted model.

Table 1.3 *Algorithm Status and Information About the Model*

Return Status	Extracted Model
`Feasible`	has been proven to be feasible. A feasible solution can be queried.
`Optimal`	has been solved to optimality. The optimal solution can be queried.
`Infeasible`	has been proven to be infeasible.
`Unbounded`	has been proven to be unbounded. The notion of unboundedness adopted by `IloCplex` does not include that the model has been proven to be feasible. Instead, what has been proven is that if there is a feasible solution with objective value x*, there exists a feasible solution with objective value x*-1 for a minimization problem, or x*+1 for a maximization problem.
`InfeasibleOrUnbounded`	has been proven to be infeasible or unbounded.
`Unknown`	has not been able to be processed far enough to prove anything about the model. A common reason may be that a time limit was hit.
`Error`	has not been able to be processed or an error occurred during the optimization.

As can be seen, these statuses indicate information about the model that the ILOG CPLEX optimizer was able to prove during the last call to method `solve()`. In addition, the

ILOG CPLEX optimizer provides information about how it terminated. For example, it may have terminated with only a feasible but not optimal solution because it hit a limit or because a user callback terminated the optimization. Further information is accessible by calling solution query routines, such as method `cplex.getCplexStatus()`, which returns a member of the nested enumeration type `IloCplex::CplexStatus`, or methods `cplex.isPrimalFeasible()` or `cplex.isDualFeasible()`.

For more information about those statuses see the *ILOG CPLEX Reference Manual*.

Querying Solution Data

If `cplex.solve()` returns `IloTrue`, a feasible solution has been found and solution values for model variables are available to be queried. For example, the solution value for the numeric variable `var1` can be accessed as follows:

```
IloNum x1 = cplex.getValue(var1);
```

However, querying solution values variable by variable may result in ugly code. Here the use of Concert Technology arrays provides a much more compact way of accessing the solution values. Assuming your variables are stored in an `IloNumVarArray var`, you can use

```
IloNumArray x(env);
cplex.getValues(x, var);
```

to access the solution values for all variables in `var` at once. Value `x[i]` contains the solution value for variable `var[i]`.

Solution data is not restricted to the solution values of variables. It also includes values of slack variables for linear constraints and the objective value. If the extracted model does not contain an objective object, `IloCplex` assumes a 0 expression objective. The objective value is returned by calling method `cplex.getObjValue()`. Slack values are accessed with the methods `getSlack()` and `getSlacks()`, which take linear constraints as a parameter.

For LPs and QPs, solution data includes information such as dual variables and reduced cost. Such information can be queried with the methods, `getDual()`, `getDuals()`, `getReducedCost()`, and `getReducedCosts()`.

Accessing Basis Information

When solving the LPs or QPs with a simplex algorithm, or when solving LPs using the barrier optimizer with crossover enabled, basis information is available as well. Basis information can be consulted using method `IloCplex::getBasisStatuses()` which returns basis status information for variables and constraints.

Such information is encoded by the nested enumeration type:

```
IloCplex::BasisStatus {
  Basic,
  AtLower,
  AtUpper,
  FreeOrSuperbasic
};
```

Performing Sensitivity Analysis

The availability of a basis for an LP allows you to perform sensitivity analysis for your model, if it is an LP. Such analysis tells you by how much you can modify your model without affecting the solution you found. The modifications supported by the sensitivity analysis function include bound changes, changes of the right hand side vector and changes of the objective function. They are analyzed by methods `IloCplex::getBoundSA()`, `IloCplex::getRHSSA()`, and `IloCplex::getObjSA()`, respectively.

Analyzing Infeasible Problems

An important feature of ILOG CPLEX is that even if no feasible solution has been found, that is, if `cplex.solve()` returns `IloFalse`, some information about the problem can be queried when solving LPs or QPs. All the methods discussed so far may successfully return information about the current (infeasible) solution which ILOG CPLEX maintains.

Unfortunately, there is no simple comprehensive rule about whether or not current solution information can be queried. This is because, by default, ILOG CPLEX uses a presolve procedure to simplify the model. If, for example, the model is proven to be infeasible during the presolve, no current solution is generated by the optimizer. If, in contrast, infeasibility is only proven by the optimizer, current solution information is available to be queried. The status returned by calling `cplex.getCplexStatus()` may help to determine which case you are facing, but it is probably safer and easier to include the methods for querying solution within try/catch statements.

When an LP has been proven to be infeasible, ILOG CPLEX provides assistance for determining the cause of the infeasibility. This is done by computing what is known as an irreducibly inconsistent set (IIS), which is a description of the minimal subproblem that is still infeasible. Here minimality is defined as being: if you remove any of the constraints (including finite bounds), the infeasibility vanishes. An IIS is computed for an infeasible LP model by calling method `cplex.getIIS()`. To compute an IIS for a QP, you should change the model to an LP by removing (zeroing) any quadratic terms from the objective function and then compute an IIS on the LP version.

Solution Quality

The ILOG CPLEX optimizer uses finite precision arithmetic to compute solutions. To compensate for numerical errors due to this, tolerances are used by which the computed solution is allowed to violate feasibility or optimality conditions. Thus the solution computed by the `solve()` method may in fact slightly violate the bounds specified in the model for example. You can call:

```
IloNum violation = cplex.getQuality(IloCplex::MaxPrimalInfeas);
```

to query the maximum bound violation among all variables and slacks. If you are also interested in the variable or constraint where the maximum violation occurs, call instead:

```
IloRange maxrange;
IloNumVar maxvar;
IloNum violation = cplex.getQuality(IloCplex::MaxPrimalInfeas,
                                    &maxrange,
                                    &maxvar);
```

ILOG CPLEX will copy the variable or constraint handle in which the maximum violation occurs to `maxvar` or `maxrange` and make the other handle an empty one. The maximum primal infeasibility is only one example of a wealth of quality measures. The full list is defined by the nested enumeration type `IloCplex::Quality`. All of these can be used as a parameter for the `getQuality()` methods, though some measures are not available for all optimizer option choices. A list of solution qualities is given in *Appendix D* of the *ILOG CPLEX 8.1 Reference Manual*.

Modifying a Model

In some applications you may want to solve the modification of another model, in order, for example, to do scenario analysis or to make adaptations based on the solution of the first model. To do this, you do not have to start a new model from scratch, but instead you can take an existing model and change it to your needs. This is done by calling the modification methods of the individual modeling objects.

When an extracted model is modified, the modification is tracked in the `cplex` object. This is done through notification. Whenever a modification method is called, `cplex` objects that have extracted the model are notified about it. The `cplex` objects then track the modification in their internal data structures.

Not only does ILOG CPLEX track all modifications of the model it has extracted, but also it tries to maintain as much solution information from a previous invocation of `solve()` as is possible and reasonable.

We already encountered what is perhaps the most important modification method, that is, the method `IloModel::add()` for adding modeling objects to a model. Conversely, you may call `IloModel::remove()` to remove a modeling object from a model. Objective functions

can be modified by changing their sense and by editing their expression, or by changing their expression completely. Similarly, the bounds of constraints and their expressions can be modified. For a complete list of supported modifications, see the documentation of the individual modeling objects in the *ILOG Concert Reference Manual*.

Deleting and Removing Modeling Objects

A special type of modification is that of deleting a modeling object by calling its `end()` method. Consider, for example, the deletion of a variable. What happens if the variable you delete has been used in constraints or the objective, or has been extracted to ILOG CPLEX? Concert Technology carefully removes the deleted variable from all other modeling objects and algorithms that may keep a reference to the variable in question. This applies to any modeling object to be removed. However, user-defined handles to the removed variable are not managed by Concert Technology. Instead it is up to the user to make sure that these handles are not used after the deletion of the modeling object. The only operation allowed then is the assignment operator.

Concert Technology also provides a way to remove a modeling object from all other modeling objects and algorithms exactly the same way as when deleting it, yet without deleting the modeling object. This is done by calling the method `removeFromAll()`. This may be helpful to temporarily remove a variable from your model while keeping the option to add it back later on.

It is important to understand the difference between the above and calling `model.remove(obj)` for an object `obj`. In this case, it does not necessarily mean that `obj` is removed from the problem ILOG CPLEX maintains. Whether or not this happens depends on the removed object being referenced by yet another extracted modeling object. Usually when a constraint is removed from the extracted model, the constraint is also removed from ILOG CPLEX as well, unless it was added to the model more than once.

Consider the case where a variable is removed from ILOG CPLEX after one of the delete or remove operations discussed above. If the `cplex` object contains a simplex basis, by default the status for that variable is removed from the basis as well. If the variable happens to be basic, the operation corrupts the basis. If this is not desired, ILOG CPLEX provides a delete mode that first pivots the variable out of the basis before removing it. The resulting basis is not guaranteed to be feasible or optimal, but it will still constitute a valid basis. To select this mode, call method:

```
cplex.setDeleteMode(IloCplex::FixBasis);
```

Similarly, when removing a constraint with the `FixBasis` delete mode, ILOG CPLEX will pivot the corresponding slack or artificial variable into the basis before removing it, to assure maintaining a valid basis. In either case, if no valid basis was available in the first place, no pivot operation is performed. To set the delete mode back to its default setting, call:

```
cplex.setDeleteMode(IloCplex::LeaveBasis);
```

Changing Variable Type

The type of a variable cannot be changed by calling modification methods. Instead, Concert Technology provides the modeling class `IloConversion`, the objects of which allow you to override the type of a variable in a model. This design allows you to use the same variable in different models with different types. Consider for example `model1` containing integer variable `x`. You can then create `model2`, as a copy of `model1`, that treats `x` as a continuous variable, with the following code:

```
IloModel model2(env);
model2.add(model1);
model2.add(IloConversion(env, x, ILOFLOAT));
```

A conversion object, that is, an instance of `IloConversion`, can only specify a type for a variable that is in a model. Converting the type more than once is an error, because there is no rule about which would have precedence. However, this is not a restriction, since you can remove the conversion from a model and add a new one.

Handling Errors

In Concert Technology two kinds of errors are distinguished:

1. Programming errors, such as:
   - accessing empty handle objects;
   - mixing modeling objects from different environments;
   - accessing Concert Technology array elements beyond an array's size; and
   - passing arrays of incompatible size to functions.

   Such errors are usually an oversight of the programmer. Once they are recognized and fixed there is usually no danger of corrupting an application. In a production version, it is not necessary to handle these kinds of errors.

   In Concert Technology such error conditions are handled using assert statements. If compiled without `-DNDEBUG`, the error check is performed and the code aborts with an error message indicating which assertion failed. A production version should then be compiled with the `-DNDEBUG` compiler option, which removes all the checking. In other words, no CPU cycles are consumed for checking the assertions.

2. Runtime errors, such as memory exhaustion.

   A correct program assumes that such failures can occur and therefore must be treated, even in a production version. In Concert Technology, if such an error condition occurs, an exception is thrown.

All exceptions thrown by Concert Technology classes (including `IloCplex`) are derived from `IloException`. Exceptions thrown by algorithm classes such as `IloCplex` are derived from its child class `IloAlgorithm::Exception`. The most common exceptions thrown by ILOG CPLEX are derived from `IloCplex::Exception`, a child class of `IloAlgorithm::Exception`.

Objects of the exception class `IloCplex::Exception` correspond to the error codes generated by the ILOG CPLEX Callable Library. The error code can be queried from a caught exception by calling method:

```
IloInt IloCplex::Exception::getStatus() const;
```

The error message can be queried by calling method:

```
const char* IloException::getMessage() const;
```

which is a virtual method inherited from the base class `IloException`. If you want to access only the message for printing to a channel or output stream, it is more convenient to use the overloaded output operator (`operator<<`) provided by Concert Technology for `IloException`.

In addition to exceptions corresponding to error codes from the C Callable Library, a `cplex` object may throw exceptions pertaining only to `IloCplex`. For example, the exception `IloCplex::MultipleObjException` is thrown if a model is extracted containing more than one objective function. Such additional exception classes are derived from class `IloCplex::Exception`; objects can be recognized by a negative status code returned when calling method `getStatus()`.

In contrast to most other Concert Technology classes, exception classes are not handle classes. Thus, the correct type of an exception is lost if it is caught by value rather than by reference (that is, using `catch(IloException& e) {...}`). This is one reason that we suggest catching `IloException` objects by reference, as demonstrated in all examples. See, for example, `ilodiet.cpp`. Some derived exceptions may carry information that would be lost if caught by value. So if you output an exception caught by reference, you may get a more precise message than when outputting the same exception caught by value.

There is a second reason for catching exceptions by reference. Some exceptions contain arrays to communicate the reason for the failure to the calling function. If this information were lost by calling the exception by value, method `end()` could not be called for such arrays and their memory would be leaked (until `env.end()` is called). After catching an exception by reference, calling the exception's method `end()` will free all the memory that may be used by arrays (or expressions) of the actual exception that was thrown.

In summary, the preferred way of catching an exception is:

```
catch (IloException& e) {
  ...
  e.end();
}
```

where `IloException` may be substituted for the desired Concert Technology exception class.

Example: Dietary Optimization

The optimization problem solved in this example is to compose a diet from a set of foods, so that the nutritional requirements are satisfied and the total cost is minimized. Example `diet.cpp` illustrates:

- Creating a Model Row by Row
- Creating a Model Column by Column
- Creating Multi-Dimensional Arrays with IloArray
- Using Arrays for Input/Output
- Solving the Model with IloCplex

Problem Representation

The problem contains a set of foods, which are the modeling variables; a set of nutritional requirements to be satisfied, which are the constraints; and an objective of minimizing the total cost of the food. There are two ways of looking at this problem:

- The problem can be modeled by rows, by entering the variables first and then adding the constraints on the variables and the objective function.
- The problem can be modeled by columns, by constructing a series of empty constraints and then inserting the variables into the constraints and the objective function.

Concert Technology is equally suited for both kinds of modeling; in fact, you can even mix both approaches in the same program. If a new food product is created, you can create a new variable for it regardless of how the model was originally built. Similarly, if a new nutrient is discovered, you can add a new constraint for it.

Creating a Model Row by Row

You walk into the store and compile a list of foods that are offered. For each food, you store the price per unit and the amount in stock. For some foods that you particularly like, you also set a minimum amount you would like to use in your diet. Then, for each of the foods, you create a modeling variable to represent the quantity to be purchased for your diet.

Now you get a medical book and look up which nutrients are known and relevant for you. For each nutrient, you note the minimum and maximum amounts that should be found in your diet. Also, you go through the list of foods and determine how much a food item will contribute for each nutrient. This gives you one constraint per nutrient, which can naturally be represented as a range constraint:

```
nutrMin[i] <= sum_j (nutrPer[i][j] * Buy[j]) <= nutrMax[i]
```

where `i` represents the number of the nutrient under consideration, `nutrMin[i]` and `nutrMax[i]` the minimum and maximum amount of nutrient `i` and `nutrPer[i][j]` the amount of nutrient `i` in food `j`. Finally, you specify your objective function:

```
minimize sum_j (cost[j] * Buy[j])
```

This way of creating the model is shown in the function `buildModelByRow`, in example `ilodiet.cpp`.

Creating a Model Column by Column

You start with the medical book where you compile the list of nutrients that you want to ensure are properly represented in your diet. For each of the nutrients, you create an empty constraint:

`nutrMin[i]` ≤ `...` ≤ `nutrMax[i]`

where `...` is left to be filled once you walk into the store. Also, you set up the objective function to minimize the cost. We refer to constraint `i` as `range[i]` and to the objective as `cost`.

Now you walk into the store and, for each food, you check the price and nutritional content. With this data you create a variable representing the amount you want to buy of the food type and install it in the objective function and constraints. That is, you create the following column:

```
cost(foodCost[j]) "+" "sum_i" (range[i](nutrPer[i][j]))
```

where the notation “+” and “`sum`” indicate that you “add” the new variable `j` to the objective `cost` and constraints range[`i`]. The value in parentheses is the linear coefficient that is used for the new variable. We chose this notation for its similarity to the syntax actually used in Concert Technology, as demonstrated in the function `buildModelByColumn`, in example `ilodiet.cpp`.

Creating Multi-Dimensional Arrays with IloArray

All data defining the problem are read from a file. The nutrients per food are stored in a two-dimensional array. Concert Technology does not provide a predefined array class; however, by using the template class `IloArray`, you can create your own two-dimensional array class. This class is defined with the type definition:

```
typedef IloArray<IloNumArray> IloNumArray2;
```

and is then ready to use, just like any predefined Concert Technology class, for example `IloNumArray`, the one-dimensional array class for numerical data.

Program Description

The main program starts by declaring the environment and terminates by calling method `end()` for the environment. The code in between is encapsulated in a try block that catches all Concert Technology exceptions and prints them to the C++ error stream `cerr`. All other exceptions are caught as well, and a simple error message is issued. The first action of the program is to evaluate command line parameters and call function `usage` in cases of misuse.

> ***Note:*** *In such cases, an exception is thrown. This ensures that* `env.end()` *is called before the program is terminated.*

Using Arrays for Input/Output

If all goes well, the input file is opened in the file `ifstream`. After that, the arrays for storing the problem data are created by declaring the appropriate variables. Then the arrays are filled by using the input operator with the data file. The data is checked for consistency and, if it fails, the program is aborted, again by throwing an exception.

After the problem data has been read and verified, we are ready to build the model. To do so we construct the model object with the declaration

```
IloModel mod(env);
```

The array `Buy` is created to store the modeling variables. Since the environment is not passed to the constructor of `Buy`, an empty handle is constructed. So at this point the variable `Buy` cannot be used.

Depending on the command line function, either `buildMethodByRow` or `buildMethodByColumn` is called. Both create the dietary model from the input data and return an array of modeling variables as an instance of the class `IloNumVarArray`. At that point, `Buy` is assigned to an initialized handle containing all the modeling variables and can be used afterwards.

Building the Model by Row

The model is created by rows using the function `buildModelByRow`. It first gets the environment from the model object passed to it. Then the modeling variables `Buy` are created. Instead of calling the constructor for the variables individually for each variable, we create the full array of variables, with the array of lower and upper bounds and the variable type as parameter. In this array, variable `Buy[i]` is created such that it has lower bound `foodMin[i]`, upper bound `foodMax[i]`, and type `type`.

The statement:

```
mod.add(IloMinimize(env, IloScalProd(Buy, foodCost)));
```

creates the objective function and adds it to the model. The `IloScalProd` function creates the expression $\sum_j$ `(Buy[j] * foodCost[j])` which is then passed to the function

IloMinimize. That function creates and returns the actual IloObjective object, which is added to the model with the call mod.add().

The following loop creates the constraints of the problem one by one and adds them to the model. First the expression $\sum_j$ (Buy[j] * nutrPer[i][j]) is created by building a Concert Technology expression. An expression variable expr of type IloExpr is created, and linear terms are added to it by using operator+= in a loop. The expression is used with the overloaded operator<= to construct a range constraint (an IloRange object) which is added to the model:

```
mod.add(nutrMin[i] <= expr <= nutrMax[i]);
```

After an expression has been used for creating a constraint, it is deleted by calling expr.end().

Finally, the array of modeling variables Buy is returned.

Building the Model by Column

The function buildModelByColumn()implements the creation of the model by columns. It begins by creating the array of modeling variables Buy of size 0. This is later populated when the columns of the problem are created and eventually returned.

The statement:

```
IloObjective cost = IloAdd(mod, IloMinimize(env));
```

creates a minimization objective function object with 0 expressions and adds it to the model. The objective object is created with the function IloMinimize. The template function IloAdd is used to add the objective object to the model and to return an objective object with the same type, so that we can store the objective in variable cost. The method IloModel::add() returns the modeling object as an IloExtractable, which cannot be assigned to a variable of a derived class such as IloObjective. Similarly an array of range constraints with 0 expressions is created, added to the model, and stored in array range.

In the following loop, the variables of the model are created one by one in columns thus, the new variables are immediately installed in the model. An IloNumColumn object col is created and initialized to describe how each new variable will be appended to the existing objective and constraints.

The IloNumColumn object col is initialized to contain the objective coefficient for the new variable. This is created with cost(foodCost[j]), that is using the overloaded operator() for IloObjective. Next, an IloNumColumn object is created for every constraint, representing the coefficient the new variable has in that constraint. Again these IloNumColumn objects are created with the overloaded operator(), this time of IloRange. The IloNumColumn objects are merged together to an aggregate IloNumColumn object using operator +=. The coefficient for row i is created with range[i](nutrPer[i][j]), which calls the overloaded operator() for IloRange objects.

When a column is completely constructed, a new variable is created for it and added to the array of modeling variables `Buy`. The construction of the variable is performed by the constructor:

```
IloNumVar(col, foodMin[j], foodMax[j], type)
```

which creates the new variable with lower bound `foodMin[j]`, upper bound `foodMax[j]` and type `type`, and adds it to the existing objective and ranges with the coefficients specified in column `col`. After creating the variable for this column, the `IloColumn` object is deleted by calling `col.end()`.

Solving the Model with IloCplex

After the model has been populated, we are ready to create the `cplex` object and extract the model to it by calling:

```
IloCplex cplex(mod);
```

It is then ready to solve the model, but for demonstration purposes we first write the extracted model to file `diet.lp`. Doing so can help you debug your model, as the file contains exactly what ILOG CPLEX sees. If it does not match what you expected, it will probably help you locate the code that generated the wrong part.

The model is then solved by calling method `solve()`. Finally, the solution status and solution vector are output to the output channel `cplex.out()`. By default this channel is initialized to `cout`. All logging during optimization is also output to this channel. To turn off logging, you would set the `out()` stream of `cplex` to a null stream by calling `cplex.setOut(env.getNullStream())`.

Complete Program

The complete program, `ilodiet.cpp`, shown here is also provided online, in the standard distribution.

Notes:

- *All the definitions needed for an ILOG CPLEX Concert Technology application are imported by including the file* `<ilcplex/ilocplex.h>`.
- *The line* `ILOSTLBEGIN` *is a macro that is needed for portability. Microsoft Visual C++ code varies, depending on whether you use the STL or not. This macro allows you to switch between both types of code without the need to otherwise change your source code.*
- *Function* `usage` *is called in case the program is executed with incorrect command line arguments.*

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilodiet.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// A dietary model.
//
// Input data:
// foodMin[j]      minimum amount of food j to use
// foodMax[j]      maximum amount of food j to use
// foodCost[j]     cost for one unit of food j
// nutrMin[i]      minimum amount of nutrient i
// nutrMax[i]      maximum amount of nutrient i
// nutrPer[i][j]   nutrition amount of nutrient i in food j
//
// Modelling variables:
// Buy[j]          amount of food j to purchase
//
// Objective:
// minimize sum(j) Buy[j] * foodCost[j]
//
// Constraints:
// forall foods i: nutrMin[i] <= sum(j) Buy[j] * nutrPer[i][j] <= nutrMax[j]
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
```

```
void usage(const char* name) {
   cerr << endl;
   cerr << "usage:   " << name << " [options] <file>" << endl;
   cerr << "options: -c  build model by column" << endl;
   cerr << "         -i  use integer variables" << endl;
   cerr << endl;
}


typedef IloArray<IloNumArray> IloNumArray2;


void
buildModelByRow(IloModel           mod,
                IloNumVarArray     Buy,
                const IloNumArray  foodMin,
                const IloNumArray  foodMax,
                const IloNumArray  foodCost,
                const IloNumArray  nutrMin,
                const IloNumArray  nutrMax,
                const IloNumArray2 nutrPer,
                IloNumVar::Type type) {
   IloEnv env = mod.getEnv();

   Buy.clear();
   IloNumVarArray tmp(env, foodMin, foodMax, type);
   Buy.add(tmp);
   tmp.end();

   IloInt i, j;
   IloInt n = foodCost.getSize();
   IloInt m = nutrMin.getSize();

   mod.add(IloMinimize(env, IloScalProd(Buy,foodCost)));
   for (i = 0; i < m; i++) {
      IloExpr expr(env);
      for (j = 0; j < n; j++) {
         expr += Buy[j] * nutrPer[i][j];
      }
      mod.add(nutrMin[i] <= expr <= nutrMax[i]);
      expr.end();
   }
}


void
buildModelByColumn(IloModel           mod,
                   IloNumVarArray     Buy,
                   const IloNumArray  foodMin,
                   const IloNumArray  foodMax,
                   const IloNumArray  foodCost,
```

```
                    const IloNumArray  nutrMin,
                    const IloNumArray  nutrMax,
                    const IloNumArray2 nutrPer,
                    IloNumVar::Type    type) {
   IloEnv env = mod.getEnv();

   IloInt i, j;
   IloInt n = foodCost.getSize();
   IloInt m = nutrMin.getSize();

   IloRangeArray range (env, nutrMin, nutrMax);
   mod.add(range);
   IloObjective cost = IloAdd(mod, IloMinimize(env));

   for (j = 0; j < n; j++) {
      IloNumColumn col = cost(foodCost[j]);
      for (i = 0; i < m; i++) {
         col += range[i](nutrPer[i][j]);
      }
      Buy.add(IloNumVar(col, foodMin[j], foodMax[j], type));
      col.end();
   }
   range.end();
}


int
main(int argc, char **argv)
{
   IloEnv env;

   try {
      const char*     filename  = "../../../examples/data/diet.dat";
      IloBool         byColumn  = IloFalse;
      IloNumVar::Type varType   = ILOFLOAT;

      IloInt i;

      for (i = 1; i < argc; ++i) {
         if (argv[i][0] == '-') {
            switch (argv[i][1]) {
            case 'c':
               byColumn = IloTrue;
               break;
            case 'i':
               varType = ILOINT;
               break;
            default:
               usage(argv[0]);
               throw (-1);
            }
         }
         else {
```

```
      filename = argv[i];
      break;
   }
}

ifstream file(filename);
if ( !file ) {
   cerr << "ERROR: could not open file '" << filename
        << "' for reading" << endl;
   usage(argv[0]);
   throw (-1);
}

// model data

IloNumArray  foodCost(env), foodMin(env), foodMax(env);
IloNumArray  nutrMin(env), nutrMax(env);
IloNumArray2 nutrPer(env);

file >> foodCost >> foodMin >> foodMax;
file >> nutrMin >> nutrMax;
file >> nutrPer;

IloInt nFoods = foodCost.getSize();
IloInt nNutr  = nutrMin.getSize();

if ( foodMin.getSize() != nFoods ||
     foodMax.getSize() != nFoods ||
     nutrPer.getSize() != nNutr  ||
     nutrMax.getSize() != nNutr    ) {
   cerr << "ERROR: Data file '" << filename
        << "' contains inconsistent data" << endl;
   throw (-1);
}

for (i = 0; i < nNutr; ++i) {
   if (nutrPer[i].getSize() != nFoods) {
      cerr << "ERROR: Data file '" << argv[0]
           << "' contains inconsistent data" << endl;
      throw (-1);
   }
}

// Build model

IloModel       mod(env);
IloNumVarArray Buy(env);
if ( byColumn ) {
   buildModelByColumn(mod, Buy, foodMin, foodMax, foodCost,
                      nutrMin, nutrMax, nutrPer, varType);
}
else {
   buildModelByRow(mod, Buy, foodMin, foodMax, foodCost,
```

```
                              nutrMin, nutrMax, nutrPer, varType);
      }

      // Solve model

      IloCplex cplex(mod);
      cplex.exportModel("diet.sav");

      cplex.solve();
      cplex.out() << "solution status = " << cplex.getStatus() << endl;

      cplex.out() << endl;
      cplex.out() << "cost   = " << cplex.getObjValue() << endl;
      for (i = 0; i < foodCost.getSize(); i++) {
         cplex.out() << "  Buy" << i << " = " << cplex.getValue(Buy[i]) << endl;
      }
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }

   env.end();

   return 0;
}

/*
cost   = 14.8557
  Buy0 = 4.38525
  Buy1 = 0
  Buy2 = 0
  Buy3 = 0
  Buy4 = 0
  Buy5 = 6.14754
  Buy6 = 0
  Buy7 = 3.42213
  Buy8 = 0
*/
```

C H A P T E R

2

ILOG Concert Technology for Java Users

In this chapter we explore the features ILOG CPLEX offers Java users to solve mathematical programming problems. An overview of the architecture is given, then techniques for creating models with ranged constraints and for creating objective functions are explained. These elements are then used to build and solve the Dietary Optimization example model, introduced in the last chapter.

A user-written application first creates an `IloCplex` object. It then uses the Concert Technology modeling interface implemented by `IloCplex` to create the variables, the constraints, and the objective function of the model to be solved. For example, every variable in a model is represented by an object that implements the Concert Technology variable interface `IloNumVar`. The user code only accesses the variable through its Concert Technology interface. Similarly, all other modeling objects are only accessed through their respective Concert Technology interfaces from the user-written application, while the actual objects are maintained in the ILOG CPLEX database.

Figure 2.1 illustrates how an application uses Concert Technology, `IloCplex` and the ILOG CPLEX database. The Java interfaces, represented by the dashed outline, do not actually consume memory. The ILOG CPLEX database includes the computing environment, its communication channels, and your problem objects.

For users familiar with object-oriented design patterns, this design is that of a factory, where `IloCplex` is a factory for modeling objects. The advantage of such a design is that code which creates a model using the Concert Technology modeling interface can be used not

only with `IloCplex`, but also with any other factory class, for instance `IloSolver`. This allows you to try different ILOG optimization technologies for solving your model.

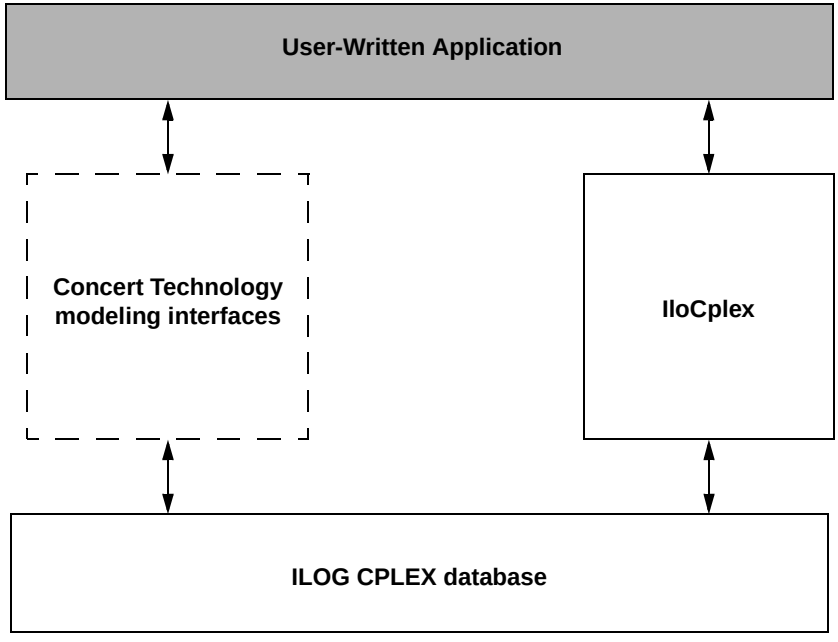

Figure 2.1 *A View of Concert Technology for Java Users*

Modeling with Concert Technology

An optimization problem is represented by a set of interconnected modeling objects in an `IloCplex` object. Modeling objects in Concert Technology are objects of type `IloNumVar` and its extensions, or `IloAddable` and its extensions. Since these are Java interfaces and not classes, objects of these types cannot be created explicitly. Rather, modeling objects are created using methods of an `IloModeler` or one of its extensions, such as `IloMPModeler` or `IloCPModeler`.

For this discussion we will concentrate on `IloModeler` and `IloMPModeler` because the class `IloCplex` implements these interfaces. To create a new modeling object, you must first create the `IloModeler` which will be used to create the modeling object. For the discussion here, the model will be an instance of `IloCplex`, and it will be created as follows:

```
IloCplex cplex = new IloCplex();
```

Since class `IloCplex` implements `IloMPModeler` (and thus its parent interface `IloModeler`) all methods from `IloMPModeler` and `IloModeler` can be used for building a model. `IloModeler` defines the methods to:

- create modeling variables of type integer, float, or Boolean;
- construct simple expressions using modeling variables;
- create objective functions; and
- create ranged constraints, that is, constraints of the form `lb` ≤`expression` ≤`ub`.

Models that consist only of such constructs can be built and solved with any ILOG optimizer implementing the `IloModeler` interface, including `IloCplex` which implements the `IloMPModeler` extension.

The `IloMPModeler` interface extends `IloModeler` by adding functionality specific to mathematical programming applications. This functionality includes these additional modeling object types:

- semi-continuous variables;
- special ordered sets; and
- piecewise linear functions.

It also includes these modeling features to support specific needs:

- change of type for previously declared variables;
- modeling by column; and
- general manipulations of model entities.

Modeling with IloModeler

`IloModeler` defines an interface for building optimization models. This interface defines methods for constructing variable, constraint, and objective function objects.

Modeling Variables

A modeling variable in Concert Technology is represented by an object of type `IloNumVar` or one of its extensions. You can choose from a variety of methods defined in `IloModeler` and `IloMPModeler` to create one or multiple modeling variable objects. An example of the method is:

```
IloNumVar x = cplex.numVar(lb, ub, IloNumVarType.Float, "xname");
```

This constructor method allows you to set all the attributes of a variable: its lower and upper bounds, its type, and its name. Names are optional in the sense that `null` strings are considered to be valid as well.

The other constructor methods for variables are provided mainly for ease of use. For example, because names are not frequently assigned to variables, all variable constructors come in pairs, where one variant requires a name string as the last parameter and the other one does not (defaulting to a `null` string).

Integer variables can be created using the `intVar()` methods, and do not require the type `IloNumVarType.Int` to be passed, as this is implied by the method name. The bound parameters are also specified more consistently as integers. These methods return objects of type `IloIntVar`, an extension of interface `IloNumVar` that allows you to query and set bounds consistently using integers, rather than doubles as used for `IloNumVar`.

Frequently, integer variables with `0/1` bounds are used as decision variables. To help create such variables, the `boolVar()` methods are provided. In the Boolean type `0` and `1` are implied, so these methods do not need to accept any bound values.

For all these constructor methods, there are also equivalent methods for creating a complete array of modeling variables at one time. These methods are called `numVarArray()`, `intVarArray()`, and `boolVarArray()`.

Building Expressions

Modeling variables are typically used in expressions that define constraints or objective functions. Expressions are represented by objects of type `IloNumExpr`. They are built using methods such as `sum()`, `prod()`, `diff()`, `negative()`, and `square()`. For example, the expression

```
x1 + 2*x2
```

where `x1` and `x2` are `IloNumVar` objects, is constructed by calling:

```
IloNumExpr expr = cplex.sum(x1, cplex.prod(2.0, x2));
```

It follows that a single variable is a special case of an expression, since `IloNumVar` is an extension of `IloNumExpr`.

The special case of linear expressions is represented by objects of type `IloLinearNumExpr`. Such expressions are editable, which is especially convenient when building up linear expressions in a loop, such as:

```
IloLinearNumExpr lin = cplex.linearNumExpr();
for (int i = 0; i < num; ++i)
  lin.addTerm(value[i], variable[i]);
```

It should be noted that the special case of the scalar product of an array of values with an array of variables is directly supported through the method `scalProd()`. Thus the above loop can be rewritten as:

```
IloLinearNumExpr lin = cplex.scalProd(value, variable);
```

It is recommended that you build expressions in terms of data that is either integer or double-precision (64 bit) floating point. Single-precision (32 bit) floating point data should

be avoided as it can result in unnecessarily ill-conditioned problems. For more information, refer to *Numerical Difficulties* on page 135.

Ranged Constraints

Ranged constraints are constraints of the form: `lb` ≤`expression` ≤`ub` and are represented in Concert Technology by objects of type `IloRange`. The most general constructor is:

```
IloRange rng = cplex.range(lb, expr, ub, name);
```

where `lb` and `ub` are double values, `expr` is of type `IloNumExpr`, and `name` is a string.

By choosing the range bounds appropriately, ranged constraints can be used to model any of the more commonly found constraints of the form:

`expr` *`rel`* `rhs`,

where *`rel`* is the relation "=", "≤", or "≥". The following table shows how to choose `lb` and `ub` for modeling these relations:

Table 2.1 *Choosing Lower and Upper Bound Values*

rel	lb	ub	method
=	rhs	rhs	eq()
≤	-Double.MAX_VALUE	rhs	le()
≥	rhs	Double.MAX_VALUE	ge()

The last column contains the method provided with `IloModeler` that can be used to directly create the appropriate ranged constraint, specifying parameters `rel` and `rhs`. For example, the constraint `expr` ≤`1.0` is created by calling

```
IloRange le = cplex.le(expr, 1.0);
```

Again, all constructors for ranged constraints come in pairs, one constructor with and one without a name parameter.

Objective Functions

The objective function in Concert Technology is represented by objects of type `IloObjective`. Such objects are defined by an optimization sense, an expression, and an optional name. The objective expression is represented by an `IloNumExpr`. The objective sense is represented by an object of class `IloObjectiveSense` and can take two values, `IloObjectiveSense.Maximize` or `IloObjectiveSense.Minimize`. The most general constructor for an objective function object is:

```
IloObjective obj = cplex.objective(sense, expr, name);
```

where `sense` is of type `IloObjectiveSense`, `expr` is of type `IloNumExpr`, and `name` is a string.

For convenience, the methods `maximize()` and `minimize()` are provided to create a maximization or minimization objective respectively, without using an `IloObjectiveSense` parameter. Names for objective function objects are optional, so all constructor methods come in pairs, one with and one without the name parameter.

The Active Model

Modeling objects, constraints and objective functions, created as described above are now added to the active model. The active model is the model implemented by the `IloCplex` object itself. In fact, `IloModeler` is an extension of the `IloModel` interface defining the model API. Thus, `IloCplex` implements `IloModel`, or in other words, an `IloCplex` object *is* a model. We will refer to the model implemented by the `IloCplex` object itself as the active model of the `IloCplex` object, or if there is no possibility of confusion between several optimizers, simply as the active model.

A model is just a set of modeling objects of type `IloAddable` such as `IloObjective` and `IloRange`. Objects of classes implementing this interface can be added to an instance of `IloModel`. Other `IloAddable` objects usable with `IloCplex` are `IloLPMatrix`, `IloConversion`, `IloSOS1`, and `IloSOS2`. These will be covered in the `IloMPModeler` section.

Variables cannot be added to a model because `IloNumVar` is not an extension of `IloAddable`. All variables used by other modeling objects (`IloAddable` objects) that have been added to a model are implicitly part of this optimization model. The explicit addition of a variable to a model can thus be avoided.

During modeling, a typical sequence of operations is to create a modeling object and immediately add it to the active model. To facilitate this for most constructor methods *methodName*`()` there is also a constructor method add*`MethodName()`* which immediately adds the newly constructed modeling object to the active model. For example, the call

```
IloObjective obj = cplex.addMaximize(expr);
```

is equivalent to

```
IloObjective obj = cplex.add(cplex.maximize(expr));
```

Not only do the add*`MethodName()`* methods simplify the program, they are also more efficient than the two equivalent calls because an intermediate copy can be avoided.

Building the Model

All the building blocks are now in place to implement a method that creates a model. The diet problem consists of finding the least expensive diet using a set of foods such that all nutritional requirements are satisfied. In this chapter, we build the specific diet model,

choose an optimizing algorithm, and see how to access more detailed information about the solution.

We have a set of foods, where food `j` has a unit cost of `foodCost[j]`. The minimum and maximum amount of food `j` which can be used in the diet is designated `foodMin[j]` and `foodMax[j]`, respectively. Each food `j` also has a nutritional value `nutrPerFood[i][j]` for all possible nutrients `i`. The nutritional requirement states that in the diet the amount of every nutrient `i` consumed must be within the bounds `nutrMin[i]` and `nutrMax[i]`.

Mathematically, this problem can be modeled using a variable `Buy[j]` for each food `j` indicating the amount of food `j` to buy for the diet. Then the objective is:

$$\text{minimize } \sum_j \texttt{(Buy[j] * foodCost[j])}$$

The nutritional requirements mean that the following conditions must be observed, that is for all `i`:

$$\texttt{nutriMin[i]} \leq \sum_i \texttt{nutrPerFood[i][j] * Buy[j]} \leq \texttt{nutriMax[i]}$$

Finally every food must be within its bounds, that is for all `j`:

$$\texttt{foodMin[j]} \leq \texttt{Buy[j]} \leq \texttt{foodMax[j]}$$

With what we have learned so far, we can implement a method that creates such a model.

```
static void buildModelByRow(IloModeler    model,
                            Data          data,
                            IloNumVar[]   Buy,
                            IloNumVarType type)
                                          throws IloException {
  int nFoods = data.nFoods;
  int nNutrs = data.nNutrs;

  for (int j = 0; j < nFoods; j++) {
    Buy[j] = model.numVar(data.foodMin[j], data.foodMax[j], type);
  }
  model.addMinimize(model.scalProd(data.foodCost, Buy));

  for (int i = 0; i < nNutrs; i++) {
    model.addRange(data.nutrMin[i],
                   model.scalProd(data.nutrPerFood[i], Buy),
                   data.nutrMax[i]);
  }
}
```

The function receives several parameters. The parameter `model` is used for two things:

- creating other modeling objects, and
- representing the model being created.

The parameter `data` contains the data for the dietary model to be built. Parameter `Buy` is an array, initialized to length `data.nFoods`, containing the model's variables. Finally, parameter `type` is used to specify the type of the variables being created.

The function starts by creating the modeling variables, one by one, and storing them in array `Buy`. Each variable `j` is initialized to have bounds `data.foodMin[j]` and `data.foodMax[j]` and to be of type `type`.

The variables are first used to construct the objective function expression with method `model.scalProd(foodCost, Buy)`. This expression is immediately used to create the minimization objective which is directly added to the active model by `addMinimize()`.

In the loop that follows, the nutritional constraints are added. For each nutrient `i` the expression representing the amount of nutrient in a diet with food levels `Buy` is computed using `model.scalProd(nutrPerFood[i], Buy)`. This amount of nutrient must be within the ranged constraint bounds `nutrMin[i]` and `nutrMax[i]`. This constraint is created and added to the active model with `addRange()`.

Note that function `buildModelByRow()` uses interface `IloModeler` rather than `IloCplex`. This allows the function to be called without change in another implementation of `IloModeler`, such as `IloSolver`.

Solving the Model

Once you have created an optimization problem in your active model, the `IloCplex` object is ready to solve it. This is done, for a model represented by `cplex` by calling:

```
cplex.solve();
```

The solve method returns a Boolean indicating whether or not a feasible solution was found and can be queried. However, when `true` is returned, the solution that was found may not be the optimal one; for example the optimization may have terminated prematurely because it ran into an iteration limit.

Additional information about a possible solution available in the `IloCplex` object can be queried with the method `getStatus()` returning an `IloCplex.Status` object. Possible statuses include:

Table 2.2 *Solution Status*

Return Status	Active Model
`Error`	It has not been possible to process the active model, or an error occurred during the optimization.
`Unknown`	It has not been possible to process the active model far enough to prove anything about it. A common reason may be that a time limit was reached.
`Feasible`	A feasible solution for the model has been proven to exist.
`Bounded`	It has been proven that the active model has a finite bound in the direction of optimization. However, this does not imply the existence of a feasible solution.
`Optimal`	The active model has been solved to optimality. The optimal solution can be queried.
`Infeasible`	The active model has been proven to possess no feasible solution.
`Unbounded`	The active model has been proven to be unbounded. The notion of unboundedness adopted by `IloCplex` is technically that of dual infeasibility; this does not include the notion that the model has been proven to be feasible. Instead, what has been proven is that if there is a feasible solution with objective value z*, there exists a feasible solution with objective value z*-1 for a minimization problem, or z*+1 for a maximization problem.
`InfeasibleOrUnbounded`	The active model has been proven to be infeasible or unbounded.

For example, an `Optimal` status indicates that an optimal solution has been found and can be queried, whereas an `Infeasible` status indicates that the active model has been proven to be infeasible. See the online *CPLEX Java Reference Manual* for more information about these statuses.

More detailed information about the status of the optimizer can be queried with method `getCplexStatus()` returning an object corresponding to ILOG CPLEX status codes. Again the online *CPLEX Java Reference Manual* contains further information about this.

Accessing Solution Information

If a solution was found with the `solve()` method, it can be accessed and then queried using a variety of methods. The objective function can be accessed by calling

```
double objval = cplex.getObjValue();
```

The values of individual modeling variables for the solution are accessed by calling methods `IloCplex.getValue()`, for example:

```
double x1 = cplex.getValue(var1);
```

Frequently, solution values for an array of variables are needed. Rather than having to implement a loop to query the solution values variable by variable, the method `IloCplex.getValues()` is provided to do so with only one function call:

```
double[] x = cplex.getValues(vars);
```

Similarly, slack values can be queried for the constraints in the active model using methods `IloCplex.getSlack()` or `IloCplex.getSlacks()`.

Printing the Solution to the Diet Model

This can now be applied to solving the diet problem we discussed earlier, and printing its solution.

```
IloCplex     cplex = new IloCplex();
IloNumVar[]  Buy   = new IloNumVar[nFoods];

if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
else            buildModelByRow   (cplex, data, Buy, varType);

// Solve model

if ( cplex.solve() ) {
  System.out.println();
  System.out.println("Solution status = " + cplex.getStatus());
  System.out.println();
  System.out.println(" cost = " + cplex.getObjValue());
  for (int i = 0; i < nFoods; i++) {
    System.out.println(" Buy" + i + " = " +
                       cplex.getValue(Buy[i]));
  }
  System.out.println();
}
```

The program extract starts by creating a new `IloCplex` object and passing it, along with the raw data in another object, either to the method `buildModelByColumn()` or `buildModelByRow()`. The array of variables returned by it is saved as array `Buy`. Then the method `solve()` is called to optimize the active model and, upon success, solution information is printed.

Choosing an Optimizer

The solution algorithm used in the `solve()` methods can be controlled and if necessary tailored to the particular needs of the model. The most important control is that of selecting the optimizer. For solving the active model, ILOG CPLEX solves one LP, one QP or a series of LPs or QPs:

- A single LP is solved if `IloCplex.isMIP()` and `IloCplex.isQuadratic()` return `false`. This is the case if the active model does not include:
  - integer variables, Boolean or semi-continuous variables;
  - special ordered sets;
  - piecewise linear functions; or
  - quadratic terms in the objective function.

`IloCplex` provides several optimizing algorithms to solve LPs.

- A single QP is solved if `IloCplex.isMIP()` returns `false` and `IloCplex.isQuadratic()` returns `true`. This is the case if the active model contains a quadratic (and positive semi-definite) objective but does not contain:
  - integer variables, Boolean or semi-continuous variables;
  - special ordered sets; or
  - piecewise linear functions.

As in the case of LPs, `IloCplex` provides several optimizing algorithms to solve QPs.

A series of LPs or QPs is solved if the active model is a MIP, which can be recognized by `IloCplex.isMIP()` returning `true`. This is the case if the model contains any of the objects excluded for single QPs above. If a MIP contains a purely linear objective function, that is `IloCplex.isQuadratic()` returns `false`, the problem is more precisely called an MILP. Otherwise it is called an MIQP. MIPs are solved using branch & cut search, which is described in more detail in Chapter 8, *Solving Mixed Integer Programming Problems*.

Solving a Single LP or QP

To solve a single LP or QP, or the first LP or QP in a series, use `IloCplex.setParam(IloCplex.IntParam.RootAlg, alg)`where `alg` is an integer

specifying the algorithm type. The following constants are defined for identifying the available algorithm types:

Table 2.3 *Algorithm Types*

alg	Algorithm Type
0	IloCplex.Algorithm.Auto
1	IloCplex.Algorithm.Primal
2	IloCplex.Algorithm.Dual
3	IloCplex.Algorithm.Network
4	IloCplex.Algorithm.Barrier
5	IloCplex.Algorithm.Sifting

If `IloCplex.setParam(IloCplex.IntParam.RootAlg, alg)` is not called, the default `IloCplex.Algorithm.Auto` will be used.

The `IloCplex.Algorithm.Sifting` algorithm is not available for QP. `IloCplex` will default to the `IloCplex.Algorithm.Auto` setting when parameter `IloCplex.IntParam.RootAlg` is set to `IloCplex.Algorithm.Sifting` for a QP.

Solving Subsequent LPs or QPs in a MIP

Parameter `IloCplex.IntParam.RootAlg` also controls the algorithm used for solving the first LP or QP when solving a MIP. The algorithm for solving all subsequent LPs or QPs is then controlled by parameter `IloCplex.IntParam.NodeAlg`. The same set of algorithm choices is available, with the exception of `IloCplex.Algorithm.Sifting` and `IloCplex.Algorithm.Network` in case of QPs.

Controlling ILOG CPLEX Optimizers

Though ILOG CPLEX defaults will prove sufficient to solve most problems, ILOG CPLEX offers a variety of other parameters to control various algorithmic choices. ILOG CPLEX parameters can take values of type `boolean`, `int`, `double`, and `string`. The parameters are accessed via parameter names defined in classes `IloCplex.BooleanParam`, `IloCplex.IntParam`, `IloCplex.DoubleParam`, and `IloCplex.StringParam` corresponding to the parameter type.

Parameters

Parameters are manipulated using `IloCplex.setParam()`. For example:

```
cplex.setParam(IloCplex.BooleanParam.PreInd, false);
```

sets the Boolean parameter `PreInd` to `false`, instructing ILOG CPLEX not to apply presolve before solving the problem.

Integer parameters often indicate a choice from a numbered list of possibilities, rather than a quantity. For example, the class `IloCplex.PrimalPricing` defines constants with the integer parameters shown in Table 2.4, *Constants in IloCplex.PrimalPricing* for better maintainability of the code.

Table 2.4 *Constants in IloCplex.PrimalPricing*

Integer Parameter	Constant in class `IloCplex.PrimalPricing`
0	`IloCplex.PrimalPricing.Auto`
1	`IloCplex.PrimalPricing.Devex`
2	`IloCplex.PrimalPricing.Steep`
3	`IloCplex.PrimalPricing.SteepQStart`
4	`IloCplex.PrimalPricing.Full`

Thus, the suggested method for setting steepest-edge pricing for use with the primal simplex algorithm is:

```
cplex.setParam(IloCplex.IntParam.PPriInd,
               IloCplex.PrimalPricing.Steep);
```

Table 2.5 gives an overview of the classes defining constants for parameters.

Table 2.5 *Classes with Parameters Defined by Integers.*

class	for use with parameters:
`IloCplex.Algorithm`	`IloCplex.IntParam.RootAlg` `IloCplex.IntParam.NodeAlg`
`IloCplex.MIPEmphasis`	`IloCplex.IntParam.MIPEmphasis`
`IloCplex.VariableSelect`	`IloCplex.IntParam.VarSel`
`IloCplex.NodeSelect`	`IloCplex.IntParam.NodeSel`
`IloCplex.DualPricing`	`IloCplex.IntParam.DPriInd`
`IloCplex.PrimalPricing`	`IloCplex.IntParam.PPriInd`

Parameters can be queried with method `IloCplex.getParam()` and reset to their default settings with method `IloCplex.setDefaults()`. The minimum and maximum value to

which an integer or double parameter can be set, is queried with methods `IloCplex.getMin()` and `IloCplex.getMax()`, respectively. The default value of a parameter is obtained with `IloCplex.getDefault()`.

When solving MIPs, another important way to control the solution process is by providing priority orders and branching directions for variables. The methods for doing so are:

- `IloCplex.setDirection()`,
- `IloCplex.setDirections()`,
- `IloCplex.setPriority()`, and
- `IloCplex.setPriorities()`.

Priority orders and branch directions allow you to control the branching performed during branch & cut in a static way.

Dynamic control of the solution process of MIPs is provided through goals or control callbacks. Goals and control callbacks are discussed for C++ in Chapter 9, *More About Using ILOG CPLEX*. The Java goals and callbacks are similar to the C++ versions. Goals and callbacks allow you to control the solution process when solving MIPs based on information generated during the solution process itself.

More Solution Information

Depending on the model being solved and the algorithm being used, more solution information is generated in addition to the objective value and solution values for variables and slacks.

Dual Solution Information

When solving an LP or QP, all the algorithms also compute dual solution information that can be queried. Reduced costs can be accessed by calling the method `IloCplex.getReducedCost()` or `IloCplex.getReducedCosts()`. Similarly, dual solution values for the ranged constraints of the active model can be queried using methods `IloCplex.getDual()` or `IloCplex.getDuals()`.

Basis Information

When solving an LP using all but `IloCplex.Algorithm.Barrier` without crossover, or when solving a QP with a Simplex optimizer, basis information is available as well. Basis information can be queried for the variables and ranged constraints of the active model using method `IloCplex.getBasisStatus()`. This method returns basis statuses for the variables or constraints using objects of type `IloCplex.BasisStatus`, with possible values::

`IloCplex.BasisStatus.Basic`,
`IloCplex.BasisStatus.AtLower`,
`IloCplex.BasisStatus.AtUpper`, and
`IloCplex.BasisStatus.FreeOrSuperbasic`.

The availability of a basis for an LP allows you to perform sensitivity analysis for your model. Such analysis tells you by how much you can modify your model without affecting the solution you found. The modifications supported by the sensitivity analysis function include variable bound changes, changes to the bounds of ranged constraints, and changes to the objective function. They are analyzed by methods `IloCplex.getBoundSA()`, `IloCplex.getRangeSA()`, `IloCplex.getRHSSA()` and `IloCplex.getObjSA()`, respectively.

Infeasible Solution Information

An important feature of ILOG CPLEX is that even if no feasible solution has been found, that is if `cplex.solve()` returns `false`, some information about the problem can be queried when solving LPs or QPs. All the methods discussed so far may successfully return information about the current (infeasible) solution ILOG CPLEX maintains.

Unfortunately, there is no simple comprehensive rule about whether or not current solution information can be queried. This is because by default, ILOG CPLEX uses a presolve procedure to simplify the model. If, for example, the model is proven to be infeasible during the presolve, no current solution is generated by the optimizer. If, in contrast, infeasibility is only proven by the optimizer, current solution information is available to be queried. The status returned by calling `cplex.getCplexStatus()` may help to determine which case you are facing, but it is probably safer and easier to include the methods for querying the solution within `try / catch` statements.

Method `IloCplex.isPrimalFeasible()` can be called to find out if a primal feasible solution has been found and can be queried. Similarly, method `IloCplex.isDualFeasible()` can be called to find out if a dual feasible solution has been found and can be queried.

When an LP has been proven to be infeasible, ILOG CPLEX provides assistance for determining the cause of the infeasibility. This is done by computing what is known as an irreducibly inconsistent set (IIS), which is a description of the minimal subproblem that is still infeasible. An infeasible subproblem is minimal if, when you remove any of the constraints (including finite bounds), the infeasibility vanishes.

An IIS is computed for an infeasible model by calling method `cplex.getIIS()`. Currently this method requires that the infeasibility was proven by the primal simplex method. It returns an object of type `IloCplex.IIS`, which can then be queried to find out what variables and constraints of the infeasible active model take part in the IIS.

Solution Quality

The ILOG CPLEX optimizer uses finite precision arithmetic to compute solutions. To compensate for numerical errors due to this, tolerances are used by which the computed solution is allowed to violate feasibility or optimality conditions. Thus the solution computed by the `solve()` method may in fact slightly violate the bounds specified in the active model.

`IloCplex` provides the method `getQuality()` to allow you to analyze the quality of the solution. Several quality measures are defined in class `IloCplex.QualityType`. For example, to query the maximal bound violation of variables or slacks of the solution found by `cplex.solve()` call:

```
IloCplex.QualityType inf =
cplex.getQuality(IloCplex.QualityType.MaxPrimalInfeas);
   double maxinfeas = inf.getValue();
```

The variable or constraint for which this maximum infeasibility occurs can be queried by calling `inf.getNumVar()` or `inf.getRange()`, one of which returns `null`. Not all quality measures are available for solutions generated by different optimizers. See the *CPLEX Java Reference Manual* for further details.

Advanced Modeling with IloMPModeler

So far we have only considered the constraints individually as ranged constraints of type `IloRange`; this is termed a *modeling by row* approach. However, mathematically the models that can be solved with `IloCplex` are frequently represented as:

$$\begin{aligned} &\min/\max\ f(x) \\ &\text{s.t. } L \le Ax \le U \\ &\qquad L \le x \le U \end{aligned}$$

where A is a *sparse matrix*. A sparse matrix is one in which a significant portion of the coefficients are zero, so algorithms and data structures can be designed to take advantage of it by storing and working with the substantially smaller subset of non-zero coefficients.

Objects of type `IloLPMatrix` are provided for use with `IloCplex` to express constraint matrices rather than individual constraints . An `IloLPMatrix` object allows you to view a set of ranged constraints and the variables used by them as a matrix, that is as:
$L \le Ax \le U$

Every row of an `IloLPMatrix` object corresponds to an `IloRange` constraint, and every column of an `IloLPMatrix` object corresponds to a modeling variable (an instance of `IloNumVar`).

An IloLPMatrix object is created with method LPMatrix() defined in IloMPModeler by calling:

```
IloLPMatrix lp = cplex.LPMatrix();
```

(or cplex.addLPMatrix() to immediately add it to the active model). The rows and columns are then added to it by specifying the non-zero matrix coefficients. Alternatively, you can add complete IloRange and IloNumVar objects to it to create new rows and columns. When adding ranged constraints, columns will be implicitly added for all the variables in the constraint expression that do not already correspond to a column of the IloLPMatrix. The IloLPMatrix object will ensure consistency of the mapping of rows to constraints and columns to variables. For example, if a ranged constraint that uses variables not yet part of the IloLPMatrix is added to the IloLPMatrix, new columns will automatically be added and associated to those variables.

See the online *ILOG CPLEX Java Reference Manual* for more information about IloLPMatrix methods.

Modeling by Column

The concept of modeling by column modeling comes from the matrix view of mathematical programming problems. Starting from a (degenerate) constraint matrix with all its rows but no columns you populate it by adding columns to it. The columns of the constraint matrix correspond to variables.

Modeling by column in ILOG CPLEX is not limited to IloLPMatrix, but can be used with IloObjective and IloRange objects as well. So for ILOG CPLEX, modeling by column can be more generally described as creating new variables and immediately installing them in existing modeling objects.

Procedure for Modeling by Column

Start by creating a description of how to install a new variable into existing modeling objects. Such a description is represented by IloColumn objects. Individual IloColumn objects describe how to install a new variable in one existing modeling object and are created with one of the IloMPModeler.column() methods. Several IloColumn objects can be linked together (with the IloCplex.add() method) to install a new variable in all modeling objects in which it is to appear. For example:

```
IloColumn col = cplex.column(obj, 1.0).and(cplex.column(rng, 2.0));
```

can be used to create a new variable and install it in the objective function represented by obj with a linear coefficient of 1.0 and in the ranged constraint rng with a linear coefficient of 2.0.

Once the proper column object has been constructed, it can be used to create a new variable by passing it as the first parameter to the variable constructor. The newly created variable will be immediately installed in existing modeling objects as described by the IloColumn object that has been used. So the line,

```
IloNumVar var = cplex.numVar(col, 0.0, 1.0);
```

creates a new variable with bounds `0.0` and `1.0` and immediately installs it in the objective `obj` with linear coefficient `1.0` and in the ranged constraint `rng` with linear coefficient `2.0`.

All constructor methods for variables come in pairs, one with and one without a first `IloColumn` parameter. Methods for constructing arrays of variables are also provided for modeling by column. These methods take an `IloColumnArray` object as parameter that describes how each individual new variable is to be installed in existing modeling objects.

Solving the Diet Model with the Java API

The problem solved in this example is to minimize the cost of a diet that satisfies certain nutritional constraints, which were described earlier in this chapter. This example was chosen because it is simple enough to be viewed from a row as well as from a column perspective. Both ways are shown in the example.

In this example either perspective can be viewed as natural. Only one approach will seem natural for many models, but there is no general way of determining which is more appropriate in a particular case.

The example accepts a filename and two options `-c` and `-i` as command line arguments. Option `-i` allows you to create a MIP model where the quantities of foods to purchase must be integers. Option `-c` can be used to build the model by columns.

The example starts by evaluating the command line arguments and reading the input data file. The input data of the dietary problem is read from a file using an object of the embedded class `Diet.Data`. Its constructor requires a file name as parameter from which it reads the data using class `InputDataReader`. This class is distributed with the examples but will not be described here as it does not make use of any ILOG CPLEX or Concert Technology functionality.

Creating the Optimizer

Once the data has been read, the `IloCplex` modeler/optimizer is created.

```
IloCplex     cplex = new IloCplex();
IloNumVar[]  Buy   = new IloNumVar[nFoods];

if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
else            buildModelByRow   (cplex, data, Buy, varType);
```

Array `IloNumVar[] Buy` is also created where the modeling variables will be stored by `buildModelByRow()` or `buildModelByColumn()`.

We have already discussed a method very similar to `buildModelByRow()`. This function is called when `byColumn` is false, which is the case when the example is executed without the `-c` command line option, otherwise, `buildModelByColumn()` is called. Note that unlike `buildModelByRow()`, this method requires `IloMPModeler` rather than `IloModeler` as parameter since modeling by column is not available with `IloModeler`.

First, the function creates an empty minimization objective and empty ranged constraints, and adds them to the active model.

```
IloObjective cost       = model.addMinimize();
IloRange[]   constraint = new IloRange[nNutrs];

for (int i = 0; i < nNutrs; i++) {
   constraint[i] = model.addRange(data.nutrMin[i], data.nutrMax[i]);
}
```

Empty means that they use a `0` expression. After that the variables are created one by one, and installed in the objective and constraints modeling by column. For each variable a column object is created. We start by creating a column object for the objective by calling:

```
IloColumn col = model.column(cost, data.foodCost[j]);
```

The column is then expanded to include the coefficients for all the constraints using `col.and()` with the column objects that are created for each constraint. This is done in the following loop:

```
for (int i = 0; i < nNutrs; i++) {
  col = col.and(model.column(constraint[i], data.nutrPerFood[i][j]));
}
```

When the full column object has been constructed it is finally used to create and install the new variable using:

```
Buy[j] = model.numVar(col, data.foodMin[j], data.foodMax[j], type);
```

Once the model is created, solving it and querying the solution is straightforward. What remains to be pointed out is the exception handling. In case of an error, ILOG CPLEX will throw an exception of type `IloException` or one of its subclasses. Thus the entire ILOG CPLEX program is enclosed in `try/catch` statements. The `InputDataReader` can throw exceptions of type `java.io.IOException` or `InputDataReader.InputDataReaderException`.

Since none of these three possible exceptions is handled elsewhere, the `main()` function ends by catching them and issuing appropriate error messages.

Example: Diet.java Source Code

Here is the entire source code listing for the example we have been presenting. It is available in the distributed files for ILOG CPLEX Java

```
// --------------------------------------------------------------------------
// File: examples/src/Diet.java
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 2001-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// A dietary model.
//
// Input data:
// foodMin[j]          minimum amount of food j to use
// foodMax[j]          maximum amount of food j to use
// foodCost[j]         cost for one unit of food j
// nutrMin[i]          minimum amount of nutrient i
// nutrMax[i]          maximum amount of nutrient i
// nutrPerFood[i][j]   nutrition amount of nutrient i in food j
//
// Modeling variables:
// Buy[j]          amount of food j to purchase
//
// Objective:
// minimize sum(j) Buy[j] * foodCost[j]
//
// Constraints:
// forall foods i: nutrMin[i] <= sum(j) Buy[j] * nutrPer[i][j] <= nutrMax[j]
//

import ilog.concert.*;
import ilog.cplex.*;


public class Diet {
   static class Data {
      int        nFoods;
      int        nNutrs;
      double[]   foodCost;
      double[]   foodMin;
      double[]   foodMax;
      double[]   nutrMin;
      double[]   nutrMax;
      double[][] nutrPerFood;

      Data(String filename) throws IloException, java.io.IOException,
                                   InputDataReader.InputDataReaderException {
```

```
      InputDataReader reader = new InputDataReader(filename);

      foodCost = reader.readDoubleArray();
      foodMin  = reader.readDoubleArray();
      foodMax  = reader.readDoubleArray();
      nutrMin  = reader.readDoubleArray();
      nutrMax  = reader.readDoubleArray();
      nutrPerFood = reader.readDoubleArrayArray();

      nFoods = foodMax.length;
      nNutrs = nutrMax.length;

      if ( nFoods != foodMin.length  ||
           nFoods != foodMax.length    )
         throw new IloException("inconsistent data in file " + filename);
      if ( nNutrs != nutrMin.length    ||
           nNutrs != nutrPerFood.length  )
         throw new IloException("inconsistent data in file " + filename);
      for (int i = 0; i < nNutrs; ++i) {
         if ( nutrPerFood[i].length != nFoods )
            throw new IloException("inconsistent data in file " + filename);
      }
   }
}

static void buildModelByRow(IloModeler    model,
                            Data          data,
                            IloNumVar[]   Buy,
                            IloNumVarType type) throws IloException {
   int nFoods = data.nFoods;
   int nNutrs = data.nNutrs;

   for (int j = 0; j < nFoods; j++) {
      Buy[j] = model.numVar(data.foodMin[j], data.foodMax[j], type);
   }
   model.addMinimize(model.scalProd(data.foodCost, Buy));

   for (int i = 0; i < nNutrs; i++) {
      model.addRange(data.nutrMin[i],
                     model.scalProd(data.nutrPerFood[i], Buy),
                     data.nutrMax[i]);
   }
}

static void buildModelByColumn(IloMPModeler  model,
                               Data          data,
                               IloNumVar[]   Buy,
                               IloNumVarType type) throws IloException {
   int nFoods = data.nFoods;
   int nNutrs = data.nNutrs;

   IloObjective cost       = model.addMinimize();
```

```
   IloRange[]   constraint = new IloRange[nNutrs];

   for (int i = 0; i < nNutrs; i++) {
      constraint[i] = model.addRange(data.nutrMin[i], data.nutrMax[i]);
   }

   for (int j = 0; j < nFoods; j++) {
      IloColumn col = model.column(cost, data.foodCost[j]);
      for (int i = 0; i < nNutrs; i++) {
         col = col.and(model.column(constraint[i], data.nutrPerFood[i][j]));
      }
      Buy[j] = model.numVar(col, data.foodMin[j], data.foodMax[j], type);
   }
}


public static void main(String[] args) {

   try {
      String          filename  = "../../../examples/data/diet.dat";
      boolean         byColumn  = false;
      IloNumVarType   varType   = IloNumVarType.Float;

      for (int i = 0; i < args.length; i++) {
         if ( args[i].charAt(0) == '-') {
            switch (args[i].charAt(1)) {
            case 'c':
               byColumn = true;
               break;
            case 'i':
               varType = IloNumVarType.Int;
               break;
            default:
               usage();
               return;
            }
         }
         else {
            filename = args[i];
            break;
         }
      }

      Data data = new Data(filename);
      int nFoods = data.nFoods;
      int nNutrs = data.nNutrs;

      // Build model
      IloCplex     cplex = new IloCplex();
      IloNumVar[]  Buy   = new IloNumVar[nFoods];

      if ( byColumn ) buildModelByColumn(cplex, data, Buy, varType);
      else            buildModelByRow   (cplex, data, Buy, varType);
```

```
            // Solve model

            if ( cplex.solve() ) {
               System.out.println();
               System.out.println("Solution status = " + cplex.getStatus());
               System.out.println();
               System.out.println(" cost = " + cplex.getObjValue());
               for (int i = 0; i < nFoods; i++) {
                  System.out.println(" Buy" + i + " = " + cplex.getValue(Buy[i]));
               }
               System.out.println();
            }
            cplex.end();
         }
         catch (IloException ex) {
            System.out.println("Concert Error: " + ex);
         }
         catch (InputDataReader.InputDataReaderException ex) {
            System.out.println("Data Error: " + ex);
         }
         catch (java.io.IOException ex) {
            System.out.println("IO Error: " + ex);
         }
      }

      static void usage() {
         System.out.println(" ");
         System.out.println("usage: java Diet [options] <data file>");
         System.out.println("options: -c  build model by column");
         System.out.println("         -i  use integer variables");
         System.out.println(" ");
      }
   }

   /*  Sample output

   Solution status = Optimal

   cost   = 14.8557
     Buy0 = 4.38525
     Buy1 = 0
     Buy2 = 0
     Buy3 = 0
     Buy4 = 0
     Buy5 = 6.14754
     Buy6 = 0
     Buy7 = 3.42213
     Buy8 = 0
   */
```

Modifying the Model

An important feature of ILOG CPLEX is that you can modify a previously created model to consider different scenarios. Furthermore, depending on the optimization model and algorithm used, ILOG CPLEX will save as much information from a previous solution as possible when optimizing a modified model.

The most important modification method is `IloModel.add()`, for adding modeling objects to the active model. Conversely, `IloModel.remove()` can be used to remove a previously added modeling object from a model.

When adding a modeling object such as a ranged constraint to a model, all the variables used by that modeling object implicitly become part of the model as well. However, when removing a modeling object, no variables are implicitly removed from the model. Instead, variables can only be explicitly removed from a model by calling `IloModel.delete()`. This will cause the specified variables to be deleted from the model, and thus from all modeling objects in the model that are using these variables. In other words, deleting variables from a model may implicitly modify other modeling objects in that model.

The API of specific modeling objects may provide modification methods. For example, variable bounds can be changed using methods `IloNumVar.setLB()` and `IloNumVar.setUB()`. Similarly the bounds of ranged constraints can be changed using `IloRange.setLB()` and `IloRange.setUB()`.

Because not all the optimizers that implement the `IloModeler` interface support the ability to modify a model, modification methods are implemented in `IloMPModeler`. These methods are for manipulating the linear expressions in ranged constraints and objective functions used with `IloCplex`. The methods `IloMPModeler.setLinearCoef()`, `IloMPModeler.setLinearCoefs()`, and `IloMPModeler.addToExpr()` apply in this situation.

The type of a variable cannot be changed. However, it can be overwritten for a particular model by adding an `IloConversion` object, which allows you to specify new types for variables within that model. When ILOG CPLEX finds a conversion object in the active model, it uses the variable types specified in the conversion object instead of the original type specified for the optimization. For example in a model containing:

```
IloNumVar x = cplex.numVar(0.0, 1.0);
cplex.add(cplex.conversion(x, IloNumVarType.Int));
```

ILOG CPLEX will only generate solutions where variable `x` is an integer (within tolerances), yet the type returned by `x.getType()` will remain `IloNumVarType.Float`.

A variable can only be used in at most one conversion object, or the model will no longer be unambiguously defined. This does not imply that the type of a variable can only be changed once and never again after that. Instead, you can remove the conversion object and add a new one to implement consecutive variable type changes.

C H A P T E R

3

The ILOG CPLEX Callable Library

This chapter describes how to write C programs using the ILOG CPLEX Callable Library. It includes sections on:

- Architecture of the ILOG CPLEX Callable Library, including information on licensing and on compiling and linking your programs
- Using the Callable Library in an Application
- ILOG CPLEX Programming Practices
- Managing Parameters from the Callable Library
- Example: Dietary Optimization

Architecture of the ILOG CPLEX Callable Library

ILOG CPLEX includes a callable C library that makes it easier to develop applications to optimize, to modify, and to interpret the results of mathematical programming problems whether linear, mixed integer, or convex quadratic ones.

You can use the Callable Library to write applications that conform to many modern computer programming paradigms, such as client-server applications within distributed computing environments, multithreaded applications running on multiple processors,

applications linked to database managers, or applications using flexible graphic user interface builders, just to name a few.

The Callable Library together with the ILOG CPLEX database make up the ILOG CPLEX *core*, as you see in Figure 3.1. The ILOG CPLEX database includes the computing environment, its communication channels, and your problem objects. You will associate the core with your application by calling library routines.

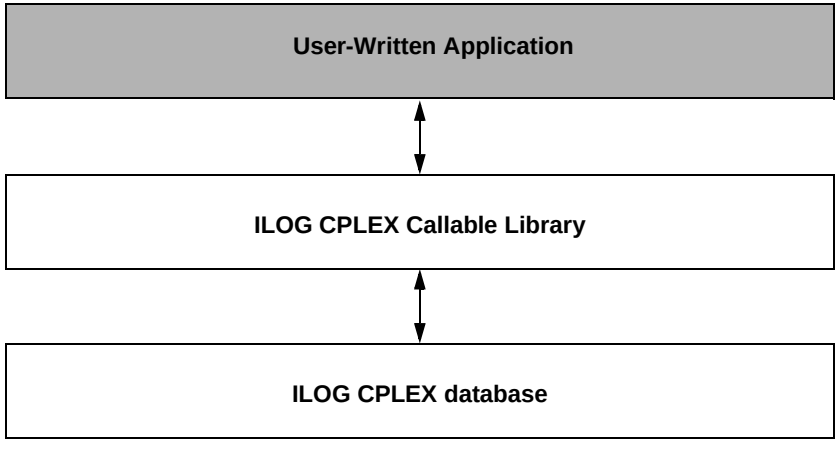

Figure 3.1 *A View of the ILOG CPLEX Callable Library*

The ILOG CPLEX Callable Library itself contains routines organized into several categories:

- *problem modification routines* let you define a problem and change it after you have created it within the ILOG CPLEX database;
- *optimization routines* enable you to optimize a problem and generate results;
- *utility routines* handle application programming issues;
- *problem query routines* access information about a problem after you have created it;
- *file reading and writing routines* move information from the file system of your operating system into your application, or from your application into the file system;
- *parameter routines* enable you to query, set, or modify parameter values maintained by ILOG CPLEX.

Licenses

ILOG CPLEX runs under the control of the ILOG License Manager (ILM). Before you can run any application program that calls ILOG CPLEX, you must have established a valid

license that it can read. Licensing instructions are provided to you separately when you buy or upgrade ILOG CPLEX. Contact your local ILOG support department if this information has not been communicated to you or if you find that you need help in establishing your ILOG CPLEX license.

Compiling and Linking

Compilation and linking instructions are provided with the files that come in the standard distribution of ILOG CPLEX for your computer platform. Check the `readme.html` file for details.

Using the Callable Library in an Application

This section tells you how to use the Callable Library in your own applications. Briefly, you must initialize the ILOG CPLEX environment, instantiate a problem object, and fill it with data. Then your application calls one of the ILOG CPLEX optimizers to optimize your problem. Optionally, your application can also modify the problem object and re-optimize it. ILOG CPLEX is designed to support this sequence of operations—modification and re-optimization of linear or quadratic programming problems (LPs or QPs)—efficiently by reusing the current *basis* of a problem as its starting point (when applicable). After it finishes using ILOG CPLEX, your application must free the problem object and release the ILOG CPLEX environment it has been using. The following sections explain these steps in greater detail.

Initialize the ILOG CPLEX Environment

ILOG CPLEX needs certain internal data structures to operate. In your own application, you use a routine from the Callable Library to initialize these data structures. You must initialize these data structures *before* your application calls any other routine in the ILOG CPLEX Callable Library.

To initialize a ILOG CPLEX environment, you must use the routine `CPXopenCPLEX()`.

This routine checks for a valid ILOG CPLEX license and then returns a C pointer to the ILOG CPLEX environment that is creates. Your application then passes this C pointer to other ILOG CPLEX routines (except `CPXmsg()`). As a developer, you decide for yourself whether the variable containing this pointer should be global or local in your application. Because the operation of checking the license can be relatively time consuming, it is strongly recommended that you call the `CPXopenCPLEX()` routine only once, or as infrequently as possible, in a program that solves a sequence of problems.

A multithreaded application needs multiple ILOG CPLEX environments. Consequently, ILOG CPLEX allows more than one environment to exist at a time.

Instantiate the Problem Object

Once you have initialized a ILOG CPLEX environment, your next step is to *instantiate* (that is, create and initialize) a *problem object* by calling `CPXcreateprob()`. This routine returns a C pointer to the problem object. Your application then passes this pointer to other routines of the Callable Library.

Most applications will use only one problem object, though ILOG CPLEX allows you to create multiple problem objects within a given ILOG CPLEX environment.

Put Data in the Problem Object

When you instantiate a problem object, it is originally empty. In other words, it has no constraints, no variables, and no coefficient matrix. ILOG CPLEX offers you several alternative ways to put data into an empty problem object (that is, to *populate* your problem object).

- You can assemble arrays of data and then call `CPXcopylp()` to copy the data into the problem object.
- You can make a sequence of calls, in any convenient order, to these routines:
  - `CPXnewcols();`
  - `CPXnewrows();`
  - `CPXaddcols();`
  - `CPXaddrows();`
  - `CPXchgcoeflist();`
- If data already exist in MPS or LP format in a file, you can call `CPXreadcopyprob()` to read that file and copy the data into the problem object. (MPS—Mathematical Programming System—is an industry-standard format for organizing data in mathematical programming problems. LP—linear programming—is an ILOG CPLEX-specific format for expressing linear programming problems as equations or inequalities. *Understanding File Formats* on page 290 explains these formats in greater detail.)
- Whenever possible, compute your problem data in double precision (64 bit). Computers are finite precision machines, and truncating your data to single precision (32 bit) can result in unnecessarily ill conditioned problems For more information, refer to *Numerical Difficulties* on page 135.

Optimize the Problem

Call one of the ILOG CPLEX optimizers to solve the problem object that you have instantiated and populated. *Choosing an Optimizer for Your LP Problem* on page 124 explains in greater detail how to choose an appropriate optimizer for your problem.

Change the Problem Object

In analyzing a given mathematical program, you may make changes in a model and study their effect. As you make such changes, you must keep ILOG CPLEX informed about the modifications so that ILOG CPLEX can efficiently re-optimize your changed problem. Always use the *problem modification routines* from the Callable Library to make such changes and thus keep ILOG CPLEX informed. In other words, do *not* change a problem by altering the original data arrays and calling `CPXcopylp()` again. That tempting strategy usually will not make the best use of ILOG CPLEX. Instead, modify your problem by means of the problem modification routines.

For example, let's say a user has already solved a given problem and then changes the upper bound on a variable by means of an appropriate call to the Callable Library. ILOG CPLEX will then begin any further optimization from the previous optimal *basis*. If that basis is still optimal with respect to the new bound, then ILOG CPLEX will return that information without even needing to refactor the basis.

Destroy the Problem Object

Use the routine `CPXfreeprob()` to destroy a problem object when your application no longer needs it.

Release the ILOG CPLEX Environment

After all the calls from your application to the ILOG CPLEX Callable Library are complete, you must release the ILOG CPLEX environment by calling the routine `CPXcloseCPLEX()`. This routine tells ILOG CPLEX that:

- all application calls to the Callable Library are complete;
- ILOG CPLEX should release any memory allocated by ILOG CPLEX for this environment;
- the application has relinquished the ILOG CPLEX license for this run, thus making the license available to the next user.

ILOG CPLEX Programming Practices

This section lists the programming practices we observe in developing and maintaining the ILOG CPLEX Callable Library.

The ILOG CPLEX Callable Library supports modern programming practices. It uses no external variables. Indeed, no global nor static variables are used in the library so that the Callable Library is fully reentrant and thread-safe. The names of all library routines begin

with the three-character prefix CPX to prevent namespace conflicts with your own routines or with other libraries. Also to avoid clutter in the namespace, there is a minimal number of routines for setting and querying parameters.

Variable Names and Calling Conventions

Routines in the ILOG CPLEX Callable Library obey the C programming convention of *call by value* (as opposed to call by reference, for example, in FORTRAN and BASIC). If a routine in the Callable Library needs the address of a variable in order to change the value of the variable, then that fact is documented in the *ILOG CPLEX Reference Manual* by the suffix _p in the parameter name in the synopsis of the routine. In C, you create such values by means of the & operator to take the address of a variable and to pass this address to the Callable Library routine.

For example, let's look at the synopses for two routines, `CPXgetobjval()` and `CPXgetx()`, as they are documented in the *ILOG CPLEX Reference Manual* to clarify this calling convention. Here is the synopsis of the routine `CPXgetobjval()`:

```
int CPXgetobjval (CPXCENVptr env, CPXCLPptr lp, double *objval_p);
```

In that routine, the third parameter is a pointer to a variable of type `double`. To call this routine from C, declare:

```
double objval;
```

Then call `CPXgetobjval()` in this way:

```
status = CPXgetobjval (env, lp, &objval);
```

In contrast, here is the synopsis of the routine `CPXgetx()`:

```
int CPXgetx (CPXENV env, CPXLPptr lp, double *x, int begin, int end);
```

You call it by creating a double-precision array by means of either one of two methods. The first method dynamically allocates the array, like this:

```
double *x = NULL;
x = (double *) malloc (100*sizeof(double));
```

The second method declares the array as a local variable, like this:

```
double x[100];
```

Then to see the optimal values for columns 5 through 104, for example, you could write this:

```
status = CPXgetx (env, lp, x, 5, 104);
```

The parameter `objval_p` in the synopsis of `CPXgetobjval()` and the parameter `x` in the synopsis of `CPXgetx()` are both of type `(double *)`. However, the suffix _p in the

parameter `objval_p` indicates that you should use an address of a single variable in one call, while the lack of `_p` in `x` indicates that you should pass an array in the other.

For guidance about how to pass values to ILOG CPLEX routines from application languages such as FORTRAN or BASIC that conventionally call by reference, see *Call by Reference* on page 97 in this manual, and consult the documentation for those languages.

Data Types

In the Callable Library, ILOG CPLEX defines a few special data types for specific ILOG CPLEX objects, as you see in Table 3.1. The types starting with `CPXC` represent the corresponding pointers to constant (`const`) objects.

Table 3.1 *Special data types in the ILOG CPLEX Callable Library*

Data type	Is a pointer to	Declaration	Set by calling
`CPXENVptr` `CPXCENVptr`	ILOG CPLEX environment	`CPXENVptr env;`	`CPXopenCPLEX()`
`CPXLPptr` `CPXCLPptr`	problem object	`CPXLPptr lp;`	`CPXcreateprob()`
`CPXNETptr` `CPXCNETptr`	problem object	`CPXNETptr net;`	`CPXNETcreateprob()`
`CPXCHANNELptr`	message channel	`CPXCHANNELptr channel;`	`CPXgetchannels()` `CPXaddchannel()`

When any of these special variables are set to a value returned by an appropriate routine, that value can be passed directly to other ILOG CPLEX routines that require such parameters. The actual internal type of these variables is a memory address (that is, a pointer); this address uniquely identifies the corresponding object. If you are programming in a language other than C, you should choose an appropriate integer type or pointer type to hold the values of these variables.

Ownership of Problem Data

The ILOG CPLEX Callable Library does not take ownership of user memory. All arguments are copied from your user-defined arrays into ILOG CPLEX-allocated memory.
ILOG CPLEX manages all problem-related memory. After you call a ILOG CPLEX routine that copies data into a ILOG CPLEX problem object, you can free the memory you used as arguments to the copying routine.

Copying in MIP and QP

The ILOG CPLEX Mixed Integer Optimizer, the routine `CPXcopyctype()` copies information about variable types in a mixed integer programming (MIP) application.

The routines `CPXcopyqsep()` and `CPXcopyquad()` are for copying information about quadratic objective coefficients in a convex quadratic programming (QP) application.

Problem Size and Memory Allocation Issues

As we indicated in *Change the Problem Object* on page 87, after you have created a problem object by calling `CPXcreateprob()`, you can modify the problem in various ways through calls to routines from the Callable Library. There is no need for you to allocate extra space in anticipation of future problem modifications. Any limit on problem size is determined by system resources and the underlying implementation of the system function `malloc()`—not by artificial limits in ILOG CPLEX.

As you modify a problem object through calls to modification routines from the Callable Library, ILOG CPLEX automatically handles memory allocations to accommodate the increasing size of the problem. In other words, you do not have to keep track of the problem size nor make corresponding memory allocations yourself as long as you are using library modification routines such as `CPXaddrows()` or `CPXaddcols()`.

However, the sequence of Callable Library routines that you invoke can influence the efficiency of memory management. Likewise, parameters controlling row growth (`CPX_PARAM_ROWGROWTH`), column growth (`CPX_PARAM_COLGROWTH`), and nonzero growth (`CPX_PARAM_NZGROWTH`) can also influence how efficiently ILOG CPLEX allocates memory to accommodate the problem object. These growth parameters determine how much extra space ILOG CPLEX allocates in its internal structures when additions to a problem object increase the size of the problem object so that it exceeds currently allocated space.

Table 3.2 *Default values of ILOG CPLEX Growth Parameters*

Parameter	Default value
CPX_PARAM_ROWGROWTH	100
CPX_PARAM_COLGROWTH	100
CPX_PARAM_NZGROWTH	500

Table 3.2 shows you the default values of these growth parameters. At these default values, if an application populates the problem object one row at a time, then ILOG CPLEX will cache the row additions until an updated problem is needed, for example when a query or optimization function is called. Similarly, it will cache column-based additions after 100 columns, and nonzero-based arrays when the additions of coefficients produces another 500

nonzeros to add to the matrix. *Memory Management and Problem Growth* on page 133 offers guidelines about performance tuning with these parameters.

Status and Return Values

The Callable Library routine `CPXopenCPLEX()` returns a pointer to a ILOG CPLEX environment. In case of failure, it returns a `NULL` pointer. The parameter `*status_p` (that is, one of its arguments) is set to `0` if the routine is successful; in case of failure, that parameter is set to a nonzero value that indicates the reason for the failure. Each failure value is unique and documented in the *ILOG CPLEX Reference Manual.*

The Callable Library routine `CPXcreateprob()` returns a pointer to a ILOG CPLEX problem object and sets its parameter `*status_p` to `0 (zero)` to indicate success. In case of failure, it returns a `NULL` pointer and sets `*status_p` to a nonzero value indicating the reason for the failure.

Some query routines in the Callable Library return a nonzero value when they are successful. For example, `CPXgetnumcols()` returns the number of columns in the constraint matrix (that is, the number of variables in the problem object). However, most query routines return `0` (zero) indicating success of the query and entail one or more parameters (such as a buffer or character string) to contain the results of the query. For example, `CPXgetrowname()` returns the name of a row in its `name` parameter.

Most other routines in the Callable Library return an integer value, `0` (zero) indicating success of the call. A nonzero return value indicates a failure. Each failure value is unique and documented in the *ILOG CPLEX Reference Manual.*

We strongly recommend that your application check the status—whether the status is indicated by the return value or by a parameter—of the routine that it calls before it proceeds.

Symbolic Constants

Most ILOG CPLEX routines return or require values that are defined as symbolic constants in the header file (that is, the include file) `cplex.h`. We highly recommend this practice of using symbolic constants, rather than hard-coded numeric values. Symbolic names improve the readability of calling applications. Moreover, if numeric values happen to change in subsequent releases of the product, the symbolic names will remain the same, thus making applications easier to maintain.

Parameter Routines

You can set many parameters in the ILOG CPLEX environment to control ILOG CPLEX operation. The values of these parameters may be integer, double, or character strings, so

there are sets of routines for accessing and setting them. Table 3.3 shows you the names and

Table 3.3 *Callable Library Routines for Parameters in the ILOG CPLEX Environment*

Type	Change value	Access current value	Access default, max, min
integer	`CPXsetintparam()`	`CPXgetintparam()`	`CPXinfointparam()`
double	`CPXsetdblparam()`	`CPXgetdblparam()`	`CPXinfodblparam()`
string	`CPXsetstrparam()`	`CPXgetstrparam()`	`CPXinfostrparam()`

purpose of these routines. Each of these routines accepts the same first argument: a pointer to the ILOG CPLEX environment (that is, the pointer returned by `CPXopenCPLEX()`). The second argument of each of those parameter routines is the parameter number, a symbolic constant defined in the header file, `cplex.h`. *Managing Parameters from the Callable Library* on page 98 offers more details about parameter settings.

Null Arguments

Certain ILOG CPLEX routines that accept optional arguments allow you to pass a `NULL` pointer in place of the optional argument. The documentation of those routines in the *ILOG CPLEX Reference Manual* indicates explicitly whether `NULL` pointer arguments are acceptable. (Passing `NULL` arguments is an effective way to avoid allocating unnecessary arrays.)

Row and Column References

Consistent with standard C programming practices, in ILOG CPLEX an array containing `k` items will contain these items in locations `0` (zero) through `k-1`. Thus a linear program with `m` rows and `n` columns will have its rows indexed from `0` to `m-1`, and its columns from `0` to `n-1`.

Within the linear programming data structure, the rows and columns that represent constraints and variables are referenced by an *index number*. Each row and column may optionally have an associated name. If you add or delete rows, the index numbers usually change:

- for deletions, ILOG CPLEX decrements each reference index above the deletion point; and
- for additions, ILOG CPLEX makes all additions at the end of the existing range.

However, ILOG CPLEX updates the names so that each row or column index will correspond to the correct row or column name. Double checking names against index numbers is the only sure way to determine which changes may have been made to matrix

indices in such a context. The routines `CPXgetrowindex()` and `CPXgetcolindex()` translate names to indices.

Character Strings

You can pass character strings as parameters to various ILOG CPLEX routines, for example, as row or column names. The Interactive Optimizer truncates output strings usually at 18 characters. Routines from the Callable Library truncate strings at 255 characters in output *text* files (such as MPS, LP, and SOS text files) but not in *binary* SAV files. Routines from the Callable Library also truncate strings at 255 characters in names that occur in messages. Routines of the Callable Library that produce log files, such as the simplex iteration log file or the MIP node log file, truncate at 16 characters. The Callable Library routine `CPXwritesol()` truncates character strings in binary solution files at 8 characters and in text solution files at 16 characters. Input, such as names read from LP and MPS files or typed interactively by the `enter` command, are truncated to 255 characters. However, we do not recommend that you rely on this truncation because unexpected behavior may result.

Checking Problem Data

If you inadvertently make an error entering problem data, the problem object will not correspond to your intentions. One possible result may be a segmentation fault or other disruption of your application. In other cases, ILOG CPLEX may solve a different model from the one you intended, and that situation may or may not result in error messages from ILOG CPLEX.

Using the Data Checking Parameter

To help you detect this kind of error, you can set the parameter `CPX_PARAM_DATACHECK` to the value `CPX_ON` to activate additional checking of array arguments for `CPXcopy...()`, `CPXread...()`, and `CPXchg...()` functions. The additional checks include:

- invalid `sense`/`ctype`/`sostype` values
- indexes out of range, for example, `rowind` ≥ `numrows`
- duplicate entries
- `matbeg` or `sosbeg` array with decreasing values
- `NAN`s in double arrays
- `NULL`s in name arrays

When the parameter is set to `CPX_OFF`, only simple checks, for example checking for the existence of the environment, are performed.

Using Diagnostic Routines for Debugging

ILOG CPLEX also provides diagnostic routines to look for common errors in the definition of problem data. In the standard distribution of ILOG CPLEX, the file `check.c` contains the source code for these routines:

- `CPXcheckcopylp()`
- `CPXcheckcopylpwnames()`
- `CPXcheckcopyqpsep()`
- `CPXcheckcopyquad()`
- `CPXcheckaddrows()`
- `CPXcheckaddcols()`
- `CPXcheckchgcoeflist()`
- `CPXcheckvals()`
- `CPXcheckcopyctype()`
- `CPXcheckcopysos()`
- `CPXNETcheckcopynet()`

Each of those routines performs a series of diagnostic tests of the problem data and issues warnings or error messages whenever it detects a potential error. To use them, you must compile and link the file `check.c`. After compiling and linking that file, you will be able to step through the source code of these routines with a debugger to help isolate problems.

If you have observed anomalies in your application, you can exploit this diagnostic capability by calling the appropriate routines just before a change or copy routine. The diagnostic routine may then detect errors in the problem data that could subsequently cause inexplicable behavior.

Those checking routines send all messages to one of the standard ILOG CPLEX message channels. You capture that output by setting the parameter `CPX_PARAM_SCRIND` (if you want messages directed to your screen) or by calling the routine `CPXsetlogfile()`.

Callbacks

The Callable Library supports callbacks so that you can define functions that will be called at crucial points in your application:

- during the presolve process;
- once per iteration in a linear programming or quadratic programming routine; and
- at various points, such as before each node is processed, in a mixed integer optimization.

In addition, callback functions can call `CPXgetcallbackinfo()` to retrieve information about the progress of an optimization algorithm. They can also return a value to indicate

whether an optimization should be aborted. `CPXgetcallbackinfo()` is the only routine of the Callable Library that a user-defined callback may call. (Of course, calls to routines not in the Callable Library are permitted.)

Using Callbacks on page 341 describes callback facilities in greater detail.

Portability

ILOG CPLEX contains a number of features to help you create Callable Library applications that can be easily ported between UNIX and Windows platforms.

CPXPUBLIC

All ILOG CPLEX Callable Library routines except `CPXmsg()` have the word `CPXPUBLIC` as part of their prototype. On UNIX platforms, this has no effect. On Win32 platforms, the `CPXPUBLIC` designation tells the compiler that all of the ILOG CPLEX functions are compiled with the Microsoft `__stdcall` calling convention. The exception `CPXmsg()` cannot be called by `__stdcall` because it takes a variable number of arguments. Consequently, `CPXmsg()` is declared as `CPXPUBVARARGS`; that calling convention is defined as `__cdecl` for Win32 systems.

Function Pointers

All ILOG CPLEX Callable Library routines that require pointers to functions expect the passed-in pointers to be declared as `CPXPUBLIC`. Consequently, when your application uses the ILOG CPLEX Callable Library routines `CPXaddfuncdest()`, `CPXsetlpcallbackfunc()`, and `CPXsetmipcallbackfunc()`, it must declare the user-written callback functions with the `CPXPUBLIC` designation. For UNIX systems, this has no effect. For Win32 systems, this will cause the callback functions to be declared with the `__stdcall` calling convention. For examples of function pointers and callbacks, see *Example: Using Callbacks* on page 351 and *Example: Using the Message Handler* on page 298.

CPXCHARptr, CPXCCHARptr, and CPXVOIDptr

The types `CPXCHARptr` and `CPXVOIDptr` are used in the header file `cplex.h` to avoid the complicated syntax of using the `CPXPUBLIC` designation on functions that return `char*`, `const char*`, or `void*`.

File Pointers

File pointer arguments for Callable Library routines should be declared with the type `CPXFILEptr`. On UNIX platforms, this practice is equivalent to using the file pointer type. On Win32 platforms, the file pointers declared this way will correspond to the environment of the ILOG CPLEX DLL. Any file pointer passed to a Callable Library routine should be obtained with a call to `CPXfopen()` and closed with `CPXfclose()`. Callable Library routines with file pointer arguments include `CPXsetlogfile()`, `CPXaddfpdest()`,

`CPXdelfpdest()`, and `CPXfputs()`. *Handling Message Channels: Callable Library Routines* on page 297 discusses most of those routines.

String Functions

Several routines in the ILOG CPLEX Callable Library make it easier to work with strings. These functions are helpful when you are writing applications in a language, such as Visual Basic, that does not allow you to dereference a pointer. The string routines in the ILOG CPLEX Callable Library are `CPXmemcpy()`, `CPXstrlen()`, `CPXstrcpy()`, and `CPXmsgstr()`.

FORTRAN Interface

The Callable Library can be interfaced with FORTRAN applications. You can download examples of a FORTRAN application from the ILOG web site at `http://www.ilog.com/products/cplex`.
Choose Tech Support; then choose Callable Library Examples. The examples were compiled on a particular platform. Since C-to-FORTRAN interfaces vary across platforms (operating system, hardware, compilers, etc.), you may need to modify the examples for your particular system.

Whether you need intermediate “glue” routines for the interface depends on your operating system. As a first step in building such an interface, we advise you to study your system documentation about C-to-FORTRAN interfaces. In that context, this section lists a few considerations particular to ILOG CPLEX in building a FORTRAN interface.

Case-Sensitivity

As you know, FORTRAN is a case-*in*sensitive language, whereas routines in the ILOG CPLEX Callable Library have names with mixed case. Most FORTRAN compilers have an option, such as the option `-U` on UNIX systems, that treats symbols in a case-sensitive way. We recommend that you use this option in any file that calls ILOG CPLEX Callable Library routines.

On some operating systems, such as DEC Alpha running Digital Unix, certain intrinsic FORTRAN functions must be in all upper case (that is, capital letters) for the compiler to accept those functions.

Underscore

On some systems, all FORTRAN external symbols are created with an underscore character (that is, _) added to the end of the symbol name. Some systems have an option to turn off this feature; for example, on DEC Alpha running Digital Unix, the option `-assume nounderscore` turns off the postpended underscore. If you are able to turn off those postpended underscores, you may not need other “glue” routines.

Six-Character Identifiers

FORTRAN 77 allows identifiers that are unique only up to six characters. However, in practice, most FORTRAN compilers allow you to exceed this limit. Since routines in the Callable Library have names greater than six characters, you need to verify whether your FORTRAN compiler enforces this limit or allows longer identifiers.

Call by Reference

By default, FORTRAN passes arguments by reference; that is, the *address* of a variable is passed to a routine, not its value. In contrast, many routines of the Callable Library require arguments passed by value. To accommodate those routines, most FORTRAN compilers have the VMS FORTRAN extension `%VAL()`. This operator used in calls to external functions or subroutines causes its argument to be passed by value (rather than by the default FORTRAN convention of passed by reference). For example, with that extension, you can call the routine `CPXprimopt()` with this FORTRAN statement:

```
status = CPXprimopt (%val(env), %val(lp))
```

Pointers

Certain ILOG CPLEX routines return a pointer to memory. In FORTRAN 77, such a pointer cannot be dereferenced; however, you can store its value in an appropriate integer type, and you can then pass it to other ILOG CPLEX routines. On most operating systems, the default integer type of four bytes is sufficient to hold pointer variables. On some systems, such as DEC Alpha, a variable of type `INTEGER*8` may be needed. Consult your system documentation to determine the appropriate integer type to hold variables that are C pointers.

Strings

When you pass strings to routines of the Callable Library, they expect C strings; that is, strings terminated by an ASCII `NULL` character, denoted `\0` in C. Consequently, when you pass a FORTRAN string, you must add a terminating `NULL` character; you do so by means of the FORTRAN intrinsic function `CHAR(0)`.

C++ Interface

The ILOG CPLEX header file, `cplex.h`, includes the `extern` C statements necessary for use with C++. If you wish to call the ILOG CPLEX C interface from a C++ application, rather than using Concert Technology, you can include `cplex.h` in your C++ source.

Managing Parameters from the Callable Library

Some ILOG CPLEX parameters assume values of type `double`; others assume values of type `int`; others are strings (that is, C-type `char*`). Consequently, in the Callable Library, there are *sets* of routines—one for `int`, one for `double`, one for `char*`—to access and to change parameters that control the ILOG CPLEX environment and guide optimization.

For example, the routine `CPXinfointparam()` shows you the default, the maximum, and the minimum values of a given parameter of type `int`, whereas the routine `CPXinfodblparam()` shows you the default, the maximum, and the minimum values of a given parameter of type `double`, and the routine `CPXinfostrparam()` shows you the default value of a given string parameter. Those three Callable Library routines observe the same conventions: they return `0` from a successful call and a nonzero value in case of error.

The routines `CPXinfointparam()` and `CPXinfodblparam()` expect five arguments:

- a pointer to the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
- an indication of the parameter to check; this argument may be a symbolic constant, such as `CPX_PARAM_CLOCKTYPE`, or a reference number, such as `1006`; the symbolic constants and reference numbers of all ILOG CPLEX parameters are documented in the *ILOG CPLEX Reference Manual* and they are defined in the include file `cplex.h`.
- a pointer to a variable to hold the default value of the parameter;
- a pointer to a variable to hold the minimum value of the parameter;
- a pointer to a variable to hold the maximum value of the parameter.

The routine `CPXinfostrparam()` differs slightly in that it does not expect pointers to variables to hold the minimum and maximum values as those concepts do not apply to a string parameter.

To access the *current* value of a parameter that interests you from the Callable Library, use the routine `CPXgetintparam()` for parameters of type `int`, `CPXgetdblparam()` for parameters of type `double`, and `CPXgetstrparam()` for string parameters. These routines also expect arguments to indicate the environment, the parameter you want to check, and a pointer to a variable to hold that current value.

No doubt you have noticed in other chapters of this manual that you can *set* parameters from the Callable Library. There are, of course, routines in the Callable Library to set such parameters: one sets parameters of type `int`; another sets parameters of type `double`; another sets string parameters.

- `CPXsetintparam()` accepts arguments to indicate:
  - the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;

- the parameter to set; this routine sets parameters of type `int`;
- the value you want the parameter to assume.

◆ `CPXsetdblparam()` accepts arguments to indicate:

- the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
- the parameter to set; this routine sets parameters of type `double`;
- the value you want the parameter to assume.

◆ `CPXsetstrparam()` accepts arguments to indicate:

- the environment; that is, a pointer of type `CPXENVptr` returned by `CPXopenCPLEX()`;
- the parameter to set; this routine sets parameters of type const `char*`;
- the value you want the parameter to assume.

The *ILOG CPLEX Reference Manual* documents the type of each parameter (`int`, `double`, `char*`) along with the symbolic constant and reference number representing the parameter.

The routine `CPXsetdefaults()` *resets all parameters* (except the log file) to their default values, including the ILOG CPLEX callback functions. This routine resets the callback functions to `NULL`. Like other Callable Library routines to manage parameters, this one accepts an argument indicating the environment, and it returns `0` for success or a nonzero value in case of error.

Example: Dietary Optimization

The optimization problem solved in this example is to compose a diet from a set of foods, so that the nutritional requirements are satisfied and the total cost is minimized. Example `diet.c` illustrates:

◆ Creating a Model Row by Row

◆ Creating a Model Column by Column

◆ Solving the Model with `CPXlpopt()`

Problem Representation

The problem contains a set of foods, which are the modeling variables; a set of nutritional requirements to be satisfied, which are the constraints; and an objective of minimizing the total cost of the food. There are two ways to look at this problem:

- The problem can be modeled in a row-wise fashion, by entering the variables first and then adding the constraints on the variables and the objective function.
- The problem can be modeled in a column-wise fashion, by constructing a series of empty constraints and then inserting the variables into the constraints and the objective function.

The diet problem is equally suited for both kinds of modeling. In fact you can even mix both approaches in the same program: If a new food product, you can create a new variable for it, regardless of how the model was originally built. Similarly, is a new nutrient is discovered, you can add a new constraint for it.

Creating a Model Row by Row

You walk into the store and compile a list of foods that are offered. For each food, you store the price per unit and the amount they have in stock. For some foods that you particularly like, you also set a minimum amount you would like to use in your diet. Then for each of the foods you create a modeling variable to represent the quantity to be purchased for your diet.

Now you get a medical book and look up which nutrients are known and relevant for you. For each nutrient, you note the minimum and maximum amount that should be found in your diet. Also, you go through the list of foods and determine how much a food item will contribute for each nutrient. This gives you one constraint per nutrient, which can naturally be represented as a range constraint

$$\texttt{nutrmin[i]} \leq \sum_{\texttt{j}} \texttt{(nutrper[i][j] * buy[j])} \leq \texttt{nutrmax[i]}$$

where `i` represents the number of the nutrient under consideration, `nutrmin[i]` and `nutrmax[i]` the minimum and maximum amount of nutrient `i` and `nutrper[i][j]` the amount of nutrient `i` in food `j`. Finally, you specify your objective function

$$\texttt{sense} = \sum_{\texttt{j}} \texttt{(cost[j] * buy[j])}$$

This way to create the model is shown in function `populatebyrow()` in example `diet.c`.

Creating a Model Column by Column

You start with the medical book where you compile the list of nutrients that you want to ensure are properly represented in your diet. For each of the nutrients you create an empty constraint:

$$\texttt{nutrmin[i]} \leq \texttt{...} \leq \texttt{nutrmax[i]}$$

where `...` is left to be filled once you walk into your store. You also set up the objective function to minimize the cost. We will refer to constraint `i` as `rng[i]` and to the objective as `cost`.

Now you walk into the store and, for each food, you check its price and nutritional content. With this data you create a variable representing the amount you want to buy of the food type and install it in the objective function and constraints. That is you create the following column:

```
cost(foodCost[j]) "+" "sum_i" (rng[i](nutrper[i][j]))
```

where the notation "+" and "sum" indicates that you “add” the new variable `j` to the objective `cost` and constraints `rng[i]`. The value in parentheses is the linear coefficient that is used for the new variable).

Program Description

All definitions needed for a ILOG CPLEX Callable Library program are imported by including file `<ilcplex/cplex.h>` at the beginning of the program. After a number of lines that establish the calling sequences for the routines that are to be used, the program’s `main()` function begins by checking for correct command line arguments, printing a usage reminder and exiting in case of errors.

Next, the data defining the problem are read from a file specified in the command line at run time. The details of this are handled in the routine `readdata()`. In this file, cost, lower bound, and upper bound are specified for each type of food; then minimum and maximum levels of several nutrients needed in the diet are specified; finally, a table giving levels of each nutrient found in each unit of food is given. The result of a successful call to this routine is two variables `nfoods` and `nnutr` containing the number of foods and nutrients in the data file, arrays `cost`, `lb`, `ub` containing the information on the foods, arrays `nutrmin`, `nutrmax` containing nutritional requirements for the proposed diet, and array `nutrper` containing the nutritional value of the foods.

Preparations to build and solve the model with ILOG CPLEX begin with the call to `CPXopenCPLEX()`. This establishes an ILOG CPLEX environment to contain the LP problem, and succeeds only if a valid ILOG CPLEX license is found.

After calls to set parameters, one to control the output that comes to the user's terminal, and another to turn on data checking for debugging purposes, a problem object is initialized through the call to `CPXcreateprob()`. This call returns a pointer to an empty problem object, which now can be populated with data.

Two alternative approaches to filling this problem object are implemented in this program, `populatebyrow()` and `populatebycolumn()`, and which one is executed is determined at run time by a calling parameter on the command line. The routine `populatebyrow()` operates by first defining all the columns through a call to `CPXnewcols()` and then repeatedly calls `CPXaddrows()` to enter the data of the constraints. The routine `populatebycolumn()` takes the complementary approach of establishing all the rows first with a call to `CPXnewrows()` and then sequentially adds the column data by calls to `CPXaddcols()`.

Solving the Model with `CPXlpopt()`

The model is at this point ready to be solved, and this is accomplished through the call to `CPXlpopt()`, which by default uses the dual simplex optimizer.

After this, the program finishes by making a call to `CPXsolution()` to obtain the values for each variable in this optimal solution, printing these values, and writing the problem to a disk file (for possible evaluation by the user) via the call to `CPXwriteprob()`. It then terminates after freeing all the arrays that have been allocated along the way.

Complete Program

The complete program, `diet.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/diet.c                                               */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* diet.c - reading data for a dietary problem, building the model
   and solving it.  methods for creating the model:

      diet -r <datafile>   generates the problem by adding rows
      diet -c <datafile>   generates the problem by adding columns
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <stdlib.h>
#include <string.h>

/* Include declaration for functions at end of program */


static int
   readarray         (FILE *in, int *num_p, double **data_p),
   readdata          (char* file,
                      int *nfoods_p, double **cost_p,
                      double **lb_p, double **ub_p,
                      int *nnutr_p, double **nutrmin_p, double **nutrmax_p,
                      double ***nutrper_p),
   populatebyrow     (CPXENVptr env, CPXLPptr lp,
                      int nfoods, double *cost, double *lb, double *ub,
                      int nnutr, double *nutrmin, double *nutrmax,
                      double **nutrper),
   populatebycolumn  (CPXENVptr env, CPXLPptr lp,
                      int nfoods, double *cost, double *lb, double *ub,
                      int nnutr, double *nutrmin, double *nutrmax,
                      double **nutrper);

static void
```

```
   free_and_null     (char **ptr),
   usage             (char *progname);


int
main (int argc, char **argv)
{
   int status = 0;

   int    nfoods;
   int    nnutr;
   double *cost     = NULL;
   double *lb       = NULL;
   double *ub       = NULL;
   double *nutrmin  = NULL;
   double *nutrmax  = NULL;
   double **nutrper = NULL;

   double *x = NULL;
   double objval;
   int    solstat;

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           i, j;

   /* Check the command line arguments */

   if (( argc != 3 )                        ||
       ( argv[1][0] != '-' )                ||
       ( strchr ("rc", argv[1][1]) == NULL )  ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   status = readdata(argv[2], &nfoods, &cost, &lb, &ub,
                     &nnutr, &nutrmin, &nutrmax, &nutrper);
   if ( status ) goto TERMINATE;


   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
```

```
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Turn on data checking */

status = CPXsetintparam (env, CPX_PARAM_DATACHECK, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on data checking, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, "diet");

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data.  For building large
   problems, consider setting the row, column and nonzero growth
   parameters before performing this task. */

switch (argv[1][1]) {
   case 'r':
      status = populatebyrow (env, lp, nfoods, cost, lb, ub,
                              nnutr, nutrmin, nutrmax, nutrper);
      break;
   case 'c':
      status = populatebycolumn (env, lp, nfoods, cost, lb, ub,
                                 nnutr, nutrmin, nutrmax, nutrper);
      break;
}

if ( status ) {
   fprintf (stderr, "Failed to populate problem.\n");
```

```
      goto TERMINATE;
   }


   /* Optimize the problem and obtain solution. */

   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   x = (double *) malloc (nfoods * sizeof(double));
   if ( x == NULL ) {
      status = CPXERR_NO_MEMORY;
      fprintf (stderr, "Could not allocate memory for solution.\n");
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, NULL, NULL, NULL);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }

   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n\n", objval);

   for (j = 0; j < nfoods; j++) {
      printf ("Food %d:  Buy = %10f\n", j, x[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "diet.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);
```

```
      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status > 0 ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   if ( nutrper != NULL ) {
      for (i = 0; i < nnutr; ++i) {
         free_and_null ((char **) &(nutrper[i]));
      }
   }
   free_and_null ((char **) &nutrper);
   free_and_null ((char **) &cost);
   free_and_null ((char **) &cost);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &nutrmin);
   free_and_null ((char **) &nutrmax);
   free_and_null ((char **) &x);

   return (status);

}  /* END main */


static int
populatebyrow (CPXENVptr env, CPXLPptr lp,
               int nfoods, double *cost, double *lb, double *ub,
               int nnutr, double *nutrmin, double *nutrmax,
               double **nutrper)
{
   int status = 0;

   int zero = 0;
   int *ind = NULL;
   int i, j;

   ind = (int*) malloc(nfoods * sizeof(int));
   if ( ind == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (j = 0; j < nfoods; j++) {
      ind[j] = j;
   }

   status = CPXnewcols (env, lp, nfoods, cost, lb, ub, NULL, NULL);
   if ( status )  goto TERMINATE;

   for (i = 0; i < nnutr; i++) {
```

```
      double rng  = nutrmax[i] - nutrmin[i];

      status = CPXaddrows (env, lp, 0, 1, nfoods, nutrmin+i, "R",
                           &zero, ind, nutrper[i], NULL, NULL);
      if ( status )  goto TERMINATE;

      status = CPXchgrngval (env, lp, 1, &i, &rng);
      if ( status )  goto TERMINATE;
   }

TERMINATE:

   free_and_null ((char **)&ind);

   return (status);

}  /* END populatebyrow */



* To populate by column, we first create the rows, and then add the
   columns.  */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp,
                  int nfoods, double *cost, double *lb, double *ub,
                  int nnutr, double *nutrmin, double *nutrmax,
                  double **nutrper)
{
   int status = 0;

   int i, j;

   int    zero    = 0;
   int    *ind    = NULL;
   double *val    = NULL;
   char   *sense  = NULL;
   double *rngval = NULL;

   sense = (char*)malloc(nnutr * sizeof(char));
   if ( sense == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (i = 0; i < nnutr; i++) {
      sense[i] = 'R';
   }

   val = (double*)malloc(nnutr * sizeof(double));
   if ( val == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   rngval = (double*)malloc(nnutr * sizeof(double));
   if ( rngval == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
```

```
   }
   for (i = 0; i < nnutr; i++) {
      rngval[i] = nutrmax[i] - nutrmin[i];
   }

   ind = (int*) malloc(nfoods * sizeof(int));
   if ( ind == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }
   for (i = 0; i < nnutr; i++) {
      ind[i] = i;
   }

   status = CPXnewrows (env, lp, nnutr, nutrmin, sense, rngval, NULL);
   if ( status )  goto TERMINATE;

   for (j = 0; j < nfoods; ++j) {
      for (i = 0; i < nnutr; i++) {
         val[i] = nutrper[i][j];
      }

      status = CPXaddcols (env, lp, 1, nnutr, cost+j, &zero,
                           ind, val, lb+j, ub+j, NULL);
      if ( status )  goto TERMINATE;
   }

TERMINATE:

   free_and_null ((char **)&sense);
   free_and_null ((char **)&rngval);
   free_and_null ((char **)&ind);
   free_and_null ((char **)&val);

   return (status);

}  /* END populatebycolumn */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */



static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s -X <datafile>\n", progname);
   fprintf (stderr,"   where X is one of the following options: \n");
   fprintf (stderr,"      r          generate problem by row\n");
```

```
   fprintf (stderr,"      c          generate problem by column\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */


static int
readarray (FILE *in, int *num_p, double **data_p)
{
   int  status = 0;
   int  max, num;
   char ch;

   num = 0;
   max = 10;

   *data_p = (double*)malloc(max * sizeof(double));
   if ( *data_p == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   for (;;) {
      fscanf (in, "%c", &ch);
      if ( ch == '\t' ||
           ch == '\r' ||
           ch == ' '  ||
           ch == '\n'   ) continue;
      if ( ch == '[' ) break;
      status = -1;
      goto TERMINATE;
   }

   for(;;) {
      int read;
      read = fscanf (in, "%lg", (*data_p)+num);
      if ( read == 0 ) {
         status = -1;
         goto TERMINATE;
      }
      num++;
      if ( num >= max ) {
         max *= 2;
         *data_p = (double*)realloc(*data_p, max * sizeof(double));
         if ( *data_p == NULL ) {
            status = CPXERR_NO_MEMORY;
            goto TERMINATE;
         }
      }
      do {
         fscanf (in, "%c", &ch);
      } while (ch == ' ' || ch == '\n' || ch == '\t'  || ch == '\r');
      if ( ch == ']' ) break;
      else if ( ch != ',' ) {
         status = -1;
         goto TERMINATE;
      }
   }
```

```
   *num_p = num;

TERMINATE:

   return (status);

} /* END readarray */


static int
readdata (char* file,
          int *nfoods_p, double **cost_p, double **lb_p, double **ub_p,
          int *nnutr_p, double **nutrmin_p, double **nutrmax_p,
          double ***nutrper_p)
{
   int status = 0;

   int ncost, nlb, nub;
   int nmin, nmax;

   int  i, n;
   char ch;
   FILE *in = NULL;

   in = fopen(file, "r");
   if ( in == NULL ) {
      status = -1;
      goto TERMINATE;
   }

   if ( (status = readarray(in, &ncost, cost_p)) ) goto TERMINATE;
   if ( (status = readarray(in, &nlb,   lb_p))   ) goto TERMINATE;
   if ( (status = readarray(in, &nub,   ub_p))   ) goto TERMINATE;
   if ( ncost != nlb  ||  ncost != nub ) {
      status = -1;
      goto TERMINATE;
   }
   *nfoods_p = ncost;

   if ( (status = readarray(in, &nmin, nutrmin_p)) ) goto TERMINATE;
   if ( (status = readarray(in, &nmax, nutrmax_p)) ) goto TERMINATE;
   if ( nmax != nmin ) {
      status = -1;
      goto TERMINATE;
   }
   *nnutr_p = nmin;

   *nutrper_p = (double**)malloc(nmin * sizeof(double*));
   if ( *nutrper_p == NULL ) {
      status = CPXERR_NO_MEMORY;
      goto TERMINATE;
   }

   for (;;) {
      fscanf (in, "%c", &ch);
      if ( ch == '\t' ||
           ch == '\r' ||
           ch == ' '  ||
```

```
            ch == '\n'   ) continue;
         if ( ch == '[' ) break;
         status = -1;
         goto TERMINATE;
      }
      for ( i = 0; i < nmin; i++ ) {
         if ( (status = readarray(in, &n, (*nutrper_p)+i)) ) goto TERMINATE;
         if ( n != ncost ) {
            status = -1;
            goto TERMINATE;
         }
         fscanf (in, "%c", &ch);
         if ( i < nmin-1  &&  ch != ',' ) {
            status = -1;
            goto TERMINATE;
         }
      }
      if ( ch != ']' ) {
         status = -1;
         goto TERMINATE;
      }


TERMINATE:

   return (status);

} /* END readdata */
```

C H A P T E R

4

Further Programming Considerations

This chapter offers suggestions for improving application development and debugging completed applications. It includes sections on:

- Tips for Successful Application Development
- Using the Interactive Optimizer for Debugging
- Eliminating Common Programming Errors

Tips for Successful Application Development

In the previous chapters, we indicated briefly the minimal steps to use the Component Libraries in an application. This section offers guidelines for successfully developing an application that exploits the ILOG CPLEX Component Libraries according to those steps. These guidelines aim to help you minimize development time and maximize application performance.

Prototype the Model

We strongly recommend that you begin by creating a small-scale version of the model for your problem. (There are modeling languages, such as ILOG OPL, that may be helpful to

you for this task.) This prototype model can serve as a test-bed for your application and a point of reference during development.

Identify Routines to Use

If you decompose your application into manageable components, you can more easily identify the tools you will need to complete the application. Part of this decomposition consists of determining which methods or routines from the ILOG CPLEX Component Libraries your application will call. Such a decomposition will assist you in testing for completeness; it may also help you isolate troublesome areas of the application during development; and it will aid you in measuring how much work is already done and how much remains.

Test Interactively

The Interactive Optimizer in ILOG CPLEX (introduced in the manual *ILOG CPLEX Getting Started*) offers a reliable means to test the ILOG CPLEX component of your application interactively, particularly if you have prototyped your model. Interactive testing through the Interactive Optimizer can also help you identify precisely which methods or routines from the Component Libraries your application needs. Additionally, interactive testing early in development may also uncover any flaws in procedural logic before they entail costly coding efforts.

Most importantly, optimization commands in the Interactive Optimizer perform exactly like optimization routines in the Component Libraries. For an LP, the `optimize` command in the Interactive Optimizer works the same way as the `cplex.solve()` and `CPXlpopt()` routines in the ILOG CPLEX Component Libraries. Consequently, any discrepancy between the Interactive Optimizer and the Component Libraries routines with respect to the solutions found, memory used, or time taken indicates a problem in the logic of the application calling the routines.

Assemble Data Efficiently

As we indicated in previous chapters, ILOG CPLEX offers several ways of putting data into your problem or (more formally) populating the problem object. You must decide which approach is best adapted to your application, based on your knowledge of the problem data and application specifications. These considerations may enter into your decision:

- If your Callable Library application builds the arrays of the problem in memory and then calls `CPXcopylp()`, it avoids time-consuming reads from disk files.
- In the Callable Library, using the routines `CPXnewcols()`, `CPXnewrows()`, `CPXaddcols()`, `CPXaddrows()`, and `CPXchgcoeflist()` may help you build modular code that will be more easily modified and maintained than code that assembles all problem data in one step.

- An application that reads an MPS or LP file may reduce the coding effort but, on the other hand, may increase runtime and disk space requirements.

  Keep in mind that if an application using the ILOG CPLEX Component Libraries reads an MPS or LP file, then some other program must generate that formatted file. The data structures used to generate the file can almost certainly be used directly to build the problem-populating arrays for `CPXcopylp()` or `CPXaddrows()`—a choice resulting in less coding and a faster, more efficient application.

  In short, formatted files are useful for prototyping your application. For production purposes, assembly of data arrays in memory may be a better enhancement.

Test Data

ILOG CPLEX provides the `DataCheck` parameter to check the correctness of data used in problem creation and problem modification methods. When this parameter is set, ILOG CPLEX will perform extra checks to determine that array arguments contain valid values, such as indices within range, no duplicate entries, valid row sense indicators and valid numerical values. These checks can be very useful during development, but are probably too costly for deployed applications. The checks are similar to but not as extensive as those performed by the `CPXcheck...()` functions provided for the C API. When the parameter is not set (the default), only simple error checks are performed, for example, checking for the existence of the environment.

Choose an Optimizer

After you have instantiated and populated a problem object, you solve it by calling one of the optimizers available in the ILOG CPLEX Component Libraries. Your choice of optimizer depends on the type of problem:

- The primal simplex, dual simplex, and primal-dual barrier methods can be used to solve linear and quadratic programs.
- The network optimizer is appropriate for solving linear and quadratic programs with large embedded networks.
- The branch & cut algorithm should be used if the problem contains discrete components (binary, integer, or semi-continuous variables, piecewise linear objective, or SOS sets).

In ILOG CPLEX, there are many possible parameter settings for each optimizer. Generally, the default parameter settings are best for linear programming and quadratic programming problems, but Chapter 5, *Solving Linear Programming Problems* and Chapter 7, *Solving Quadratic Programming Problems* offer more detail about improving performance with respect to these problems. Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them, as suggested in Chapter 8, *Solving Mixed Integer Programming Problems*.

In either case, the Interactive Optimizer in ILOG CPLEX lets you try different parameter settings and different optimizers to determine the best optimization procedure for your particular application. From what you learn by experimenting with commands in the Interactive Optimizer, you can more readily choose which method or routine from the Component Libraries to call in your application.

Program with a View toward Maintenance and Modifications

Good programming practices save development time and make an application easier to understand and modify. *Tips for Successful Application Development* on page 113 describes our programming conventions in developing ILOG CPLEX. In addition, we recommend the following programming practices.

Comment Your Code

Comments, written in mixed upper- and lower-case, will prove useful to you at a later date when you stare at code written months ago and try to figure out what it does. They will also prove useful to our staff, should you need to send us your application for technical support.

Write Readable Code

Follow conventional formatting practices so that your code will be easier to read, both for you and for others. Use fewer than 80 characters per line. Put each statement on a separate line. Use white space (for example, space, blank lines, tabs) to distinguish logical blocks of code. Display compound loops with clearly indented bodies. Display `if` statements like combs; that is, we recommend that you align `if` and `else` in the same column and then indent the corresponding block. Likewise, we recommend that you indent the body of compound statements, loops, and other structures distinctly from their corresponding headers and closing brackets. Use uniform indentation (for example, three to five spaces). Put at least one space before and after each relational operator, as well as before and after each binary plus (`+`) and minus (`-`). Use space as you do in normal English.

Avoid Side-Effects

We recommend that you minimize side-effects by avoiding expressions that produce internal effects. In C, for example, try to avoid expressions of this form:

```
a = c + (d =  e*f);  /* A BAD IDEA */
```

where the expression assigns the values of `d` and `a`.

Don’t Change Argument Values

A user-defined function should not change the values of its arguments. Do not use an argument to a function on the left-hand side of an assignment statement in that function. Since C and C++ pass arguments by value, treat the arguments strictly as values; do not change them inside a function.

Declare the Type of Return Values

Always declare the return type of functions explicitly. Though C has a “historical tradition” of making the default return type of all functions `int`, we recommend that you explicitly declare the return type of functions that return a value, and use `void` for procedures that do not return a value.

Manage the Flow of Your Code

Use only one `return` statement in any function. Limit your use of `break` statements to the inside of `switch` statements. In C, do not use `continue` statements and limit your use of `goto` statements to exit conditions that branch to the end of a function. Handle error conditions in C++ with a `try`/`catch` block and in C with a `goto` statement that transfers control to the end of the function so that your functions have only one exit point.

In other words, control the flow of your functions so that each block has one entry point and one exit point. This “one way in, one way out” rule makes code easier to read and debug.

Localize Variables

Avoid global variables at all costs. Code that exploits global variables invariably produces side-effects which in turn make the code harder to debug. Global variables also set up peculiar reactions that make it difficult to include your code successfully within other applications. Also global variables preclude multithreading unless you invoke locking techniques. As an alternative to global variables, pass arguments down from one function to another.

Name Your Constants

Scalars—both numbers and characters—that remain constant throughout your application should be named. For example, if your application includes a value such as 1000, create a constant with the `#define` statement to name it. If the value ever changes in the future, its occurrences will be easy to find and modify as a named constant.

Choose Clarity First, Efficiency Later

Code first for clarity. Get your code working accurately first so that you maintain a good understanding of what it is doing. Then, once it works correctly, look for opportunities to improve performance.

Debug Effectively

Using Diagnostic Routines for Debugging on page 94, contains tips and guidelines for debugging an application that uses the ILOG CPLEX Callable Library. In that context, we recommend using a symbolic debugger as well as other widely available development tools to produce error-free code.

Test Correctness, Test Performance

Even a program that has been carefully debugged so that it runs correctly may still contain errors or “features” that inhibit its performance with respect to execution speed, memory

use, and so forth. Just as the ILOG CPLEX Interactive Optimizer can aid in your tests for correctness, it can also help you improve performance. It uses the same routines as the Component Libraries; consequently, it requires the same amount of time to solve a problem created by a callable-library application. We recommend that you use `CPXwriteprob()`, specifying a file type of SAV, to create a binary representation of the problem object of your application. Then read that representation into the Interactive Optimizer, and solve it there. If your application sets parameters, use the same settings in the Interactive Optimizer. If you find that your application takes significantly longer to solve the problem than does the Interactive Optimizer, then you can probably improve the performance of your application. In such a case, look closely at issues like memory fragmentation, unnecessary compiler options, inappropriate linker options, and programming practices that slow the application without causing incorrect results (such as operations within a loop that should be outside the loop).

Using the Interactive Optimizer for Debugging

The ILOG CPLEX Interactive Optimizer distributed with the Component Libraries offers a way to see what is going on within the ILOG CPLEX part of your application when you observe peculiar behavior in your optimization application. The commands of the Interactive Optimizer correspond exactly to routines of the Component Libraries, so anomalies due to the ILOG CPLEX-part of your application will manifest themselves in the Interactive Optimizer as well, and contrariwise, if the Interactive Optimizer behaves appropriately on your problem, you can be reasonably sure that routines you call in your application from the Component Libraries work in the same appropriate way.

The first step in using the Interactive Optimizer for debugging is to write a version of the problem from the application into a formatted file that can then be loaded into the Interactive Optimizer. To do so, insert a call to the method `IloCplex::exportModel()` or to the routine `CPXwriteprob()` into your application. Use that call to create a file, whether an LP, SAV, or MPS formatted problem file. (*Understanding File Formats* on page 290 briefly describes these file formats.) Then read that file into the Interactive Optimizer and optimize the problem there.

Note that MPS, LP and SAV files have differences that influence how to interpret the results of the Interactive Optimizer for debugging. SAV files contain the exact binary representation of the problem as it appears in your program, while MPS and LP files are text files containing possibly less precision for numeric data. And, unless every variable appears on the objective function, ILOG CPLEX will probably order the variables differently when it reads the problem from an LP file than from an MPS or SAV file. With this in mind, SAV files are the most useful for debugging using the Interactive Optimizer, followed by MPS files, then finally LP files, in terms of the change in behavior you might see by use of explicit files. On the other hand, LP files are often quite helpful when you want to examine the problem, more so than as input for the Interactive Optimizer. Furthermore, try solving

both the SAV and MPS files of the same problem using the Interactive Optimizer. Different results may provide additional insight into the source of the difficulty. In particular, use the following guidelines with respect to reproducing your program's behavior in the Interactive Optimizer.

1. If you can reproduce the behavior with a SAV file, but not with an MPS file, this suggests corruption or errors in the problem data arrays. Use the `DataCheck` parameter or diagnostic routines in the source file `check.c` to track down the problem.

2. If you can reproduce the behavior in neither the SAV file nor the MPS file, the most likely cause of the problem is that your program has some sort of memory error. Memory debugging tools such as Purify or Insure will usually find such problems quickly.

3. When solving a problem in MPS or LP format, if the Interactive Optimizer issues a message about a segmentation fault or similar ungraceful interruption and exits, contact ILOG CPLEX technical support to arrange for transferring the problem file. The Interactive Optimizer should never exit with a system interrupt when solving a problem from a text file, even if the program that created the file has errors. Such cases are extremely rare.

If the peculiar behavior that you observed in your application persists in the Interactive Optimizer, then you must examine the LP or MPS or SAV problem file to determine whether the problem file actually defines the problem you intended. If it does not define the problem you intended to optimize, then the problem is being passed incorrectly from your application to ILOG CPLEX, so you need to look at that part of your application.

Make sure the problem statistics and matrix coefficients indicated by the Interactive Optimizer match the ones for the intended model in your application. Use the Interactive Optimizer command `display problem stats` to verify that the size of the problem, the sense of the constraints, and the types of variables match your expectations. For example, if your model is supposed to contain only general integer variables, but the Interactive Optimizer indicates the presence of binary variables, check the type variable passed to the constructor of the variable (Concert Technology Library) or check the specification of the `ctype` array and the routine `CPXcopyctype()` (Callable Library). You can also examine the matrix, objective, and right-hand side coefficients in an LP or MPS file to see if they are consistent with the values you expect in the model.

Eliminating Common Programming Errors

We hope this section serves as a checklist to help you eliminate common pitfalls from your application.

Check Your Include Files

Make sure that the header file `ilocplex.h` (Concert Technology Library) or `cplex.h` (Callable Library) is included at the top of your application source file. If that file is not included, then compile-time, linking, or runtime errors may occur.

Clean House and Try Again

Remove all object files, recompile, and relink your application.

Read Your Messages

ILOG CPLEX detects many different kinds of errors and generates exception, warnings, or error messages about them.

To query exceptions in the Concert Technology Library, use the methods:

```
IloInt IloCplex::Exception::getStatus() const;
const char* IloException::getMessage() const;
```

To view warnings and error messages in the Callable Library, you must direct them either to your screen or to a log file.

- To direct all messages to your screen, use the routine `CPXsetintparam()` to set the parameter `CPX_PARAM_SCRIND`.
- To direct all messages to a log file, use the routine `CPXsetlogfile()`.

Check Return Values

Most methods and routines of the Component Libraries return a value that indicates whether the routine failed, where it failed, and why it failed. This return value can help you isolate the point in your application where an error occurs.

If a return value indicates failure, always check whether sufficient memory is available.

Beware of Numbering Conventions

If you delete a portion of a problem, ILOG CPLEX changes not only the dimensions but also the indices of the problem. If your application continues to use the former dimensions and indices, errors will occur. Therefore, in parts of your application that delete portions of the problem, look carefully at how dimensions and indices are represented.

Make Local Variables Temporarily Global

If you are having difficulty tracking down the source of an anomaly in the heap, try making certain local variables *temporarily* global. This debugging trick may prove useful after your

application reads in a problem file or modifies a problem object. If application behavior changes when you change a local variable to global, then you may get from it a better idea of the source of the anomaly.

Solve the Problem You Intended

Your application may inadvertently alter the problem and thus produce unexpected results. To check whether your application is solving the problem you intended, use the Interactive Optimizer, as we suggest on page 118, and the diagnostic routines, as described on page 94.

You should not ignore any ILOG CPLEX warning message in this situation either, so read your messages, as we suggest on page 120.

If you are working in the Interactive Optimizer, we also suggest that you use the command `display problem stats` to check the problem dimensions.

Special Considerations for Fortran

Check row and column indices. Fortran conventionally numbers from one (`1`), whereas C and C++ number from zero (`0`). This difference in numbering conventions can lead to unexpected results with regard to row and column indices when your application modifies a problem or exercises query routines.

We strongly recommend that you use the Fortran declaration `IMPLICIT NONE` to help you detect any unintended type conversions, as such inadvertent conversions frequently lead to strange application behavior.

Tell Us

Finally, if your problem remains unsolved by ILOG CPLEX, or if you believe you have discovered a bug in ILOG CPLEX, we would appreciate hearing from you about it.

C H A P T E R

5

Solving Linear Programming Problems

The preceding chapters have focused on the details of writing applications that model optimization problems and access the solutions to those problems, with minimal attention to the optimizer that solves them. This is because most models are solved well by the default optimizers provided by ILOG CPLEX. For instances where a user wishes to exert more direct influence over the solution process, ILOG CPLEX provides a number of features that may be of interest.

This chapter tells you more about solving linear programs with ILOG CPLEX using the LP optimizers. It contains sections on:

- Choosing an Optimizer for Your LP Problem
- Tuning LP Performance
- Diagnosing Performance Problems
- Diagnosing LP Infeasibility
- Example: Using a Starting Basis in an LP Problem
- Solving LP Problems with the Barrier Optimizer
- Interpreting the Barrier Log File
- Understanding Solution Quality from the Barrier LP Optimizer
- Overcoming Numerical Difficulties
- Diagnosing Barrier Optimizer Infeasibility

Choosing an Optimizer for Your LP Problem

ILOG CPLEX offers several different optimizers for linear programming problems. Each of these optimizers is available whether you call ILOG CPLEX from within your own application using Concert or the Callable Library, or you use the Interactive Optimizer.

The choice of LP optimizer in ILOG CPLEX can be specified through a parameter, named `CPX_PARAM_LPMETHOD` in the Callable Library, and named `LPMETHOD` in the Interactive Optimizer. In both the C++ and Java versions of Concert, the LP method is controlled by the `RootAlg` parameter (which also controls related aspects of QP and MIP solutions, as explained in the corresponding chapters of this manual). In this chapter this parameter will be referred to uniformly as `LPMethod`.

When you solve a model via:

- `cplex.solve()` (Concert)
- `CPXlpopt()` (Callable Library)
- `optimize` (Interactive Optimizer)

the `LPMethod` parameter determines which optimizer will be used.

The choices for `LPMethod`, which are explained in the next subsections, are:

Table 5.1 *LPMethod Parameter Values for Optimizer Choice*

LPMethod Values	Meaning
0	Automatic (default setting)
1	Primal Simplex
2	Dual Simplex
3	Network Simplex
4	Barrier
5	Sifting
6	Concurrent Dual, Barrier, and Primal

and the symbolic names for these in an application program are as follows:

Table 5.2 *Symbolic Names for LP Solution Methods*

	Concert — C++	Concert — Java	Callable Library
0	`IloCplex::AutoAlg`	`IloCplex.Algorithm.Auto`	`CPX_ALG_AUTOMATIC`
1	`IloCplex::Primal`	`IloCplex.Algorithm.Primal`	`CPX_ALG_PRIMAL`
2	`IloCplex::Dual`	`IloCplex.Algorithm.Dual`	`CPX_ALG_DUAL`
3	`IloCplex::Network`	`IloCplex.Algorithm.Network`	`CPX_ALG_NET`
4	`IloCplex::Barrier`	`IloCplex.Algorithm.Barrier`	`CPX_ALG_BARRIER`
5	`IloCplex::Sifting`	`IloCplex.Algorithm.Sifting`	`CPX_ALG_SIFTING`
6	`IloCplex::Concurrent`	`IloCplex.Algorithm.Concurrent`	`CPX_ALG_CONCURRENT`

Automatic Selection of Optimizer

The default `Automatic` setting of `LPMethod` lets ILOG CPLEX determine the algorithm to use to optimize your problem. Most models are solved well with this setting, and we recommend using this option except when you have a compelling reason to tune performance for a particular class of model.

On a serial computer, or on a parallel computer where only one thread will be invoked, the automatic setting will in most cases choose the dual simplex optimizer. An exception to this rule is when an advanced basis is present that is ascertained to be primal feasible; in that case, primal simplex will be called.

On a computer where parallel threads are available to ILOG CPLEX, the automatic setting results in the Concurrent optimizer being called. An exception is when an advanced basis is present; in that case, it will behave as the serial algorithm would.

Dual Simplex Optimizer

If you are familiar with linear programming theory, then you recall that a linear programming problem can be stated in primal or dual form, and an optimal solution (if one exists) of the dual has a direct relationship to an optimal solution of the primal model. ILOG CPLEX's Dual Simplex Optimizer makes use of this relationship, but still reports the solution in terms of the primal model. Recent computational advances in the dual simplex method have made it the first choice for optimizing a linear programming problem. This is especially true for primal-degenerate problems with little variability in the right-hand side coefficients but significant variability in the cost coefficients.

Primal Simplex Optimizer

ILOG CPLEX's Primal Simplex Optimizer also can effectively solve a wide variety of linear programming problems with its default parameter settings. With recent advances in the dual simplex method, the primal simplex method is no longer the obvious choice for a first try at

optimizing a linear programming problem. However, this method will sometimes work better on problems where the number of variables exceeds the number of constraints significantly, or on problems that exhibit little variability in the cost coefficients. Few problems exhibit poor numerical performance in both primal and dual form. Consequently, if you have a problem where numerical difficulties occur when you use the dual simplex optimizer, then consider using the primal simplex optimizer instead.

Network Optimizer

If a major part of your problem is structured as a network, then the ILOG CPLEX Network Optimizer may have a positive impact on performance. The ILOG CPLEX Network Optimizer recognizes a special class of linear programming problems with network structure. It uses highly efficient network algorithms on that part of the problem to find a solution from which it then constructs an advanced basis for the rest of your problem. From this advanced basis, ILOG CPLEX then iterates to find a solution to the full problem. Chapter 6, *Solving Network-Flow Problems* describes this optimizer in greater detail.

Barrier Optimizer

The barrier optimizer offers an approach particularly efficient on large, sparse problems (for example, more than 1000 rows or columns, and no more than perhaps a dozen nonzeros per column). The barrier optimizer is sufficiently different in nature from the other optimizers that it is discussed in detail at the end of this chapter, in *Solving LP Problems with the Barrier Optimizer* on page 156.

Sifting Optimizer

Sifting was developed to exploit the characteristics of models with large aspect ratios (that is, a large ratio of the number of columns to the number or rows). In particular, the method is well suited to large aspect ratio models where an optimal solution can be expected to place most variables at their lower bounds. The sifting algorithm can be thought of as an extension to the familiar simplex method. It starts by solving a subproblem, called the working problem, consisting of all rows but only a small subset of the full set of columns, by assuming an arbitrary value (such as its lower bound) for the solution value of each of the remaining columns. This solution is then used to re-evaluate the reduced costs of the remaining columns. Any columns whose reduced costs violate the optimality criterion become candidates to be added to the working problem for the next major sifting iteration. When no candidates are present, the solution of the working problem is optimal to the full problem, and sifting terminates.

The choice of optimizer to solve the working problem is governed by the `SiftAlg` parameter. You can set this parameter to any of the values accepted by the `LPMethod` parameter, except for `Concurrent` and of course `Sifting` itself. Under the default

SiftAlg setting, ILOG CPLEX chooses the optimizer automatically, typically switching between primal simplex and barrier as the optimization proceeds.

Concurrent Optimizer

On a multiprocessor computer, the concurrent optimizer launches distinct LP optimizers on multiple threads, terminating as soon as the first optimizer finishes. The first thread uses the same strategy as the single-processor "automatic" LPMethod setting (0). If a second thread is available, the concurrent optimizer runs the barrier optimizer on it. If a third processor is available, dual simplex, primal simplex, and barrier are all run. All further available threads are devoted to parallelizing the barrier optimization. It should be noted that the barrier optimization is not considered complete until the crossover step has been performed and simplex reoptimization has converged; in other words, regardless of which optimizer turns out to be the fastest, the concurrent optimizer always returns a basic solution at optimality.

The concurrent optimizer requires more memory than any individual optimizer, and of course it adds system load by consuming more aggregate CPU time than the fastest individual optimizer would alone. But the advantages offered in terms of robust solution of models, and assurance in most cases of the minimum solution time, will make it attractive in many situations.

Tuning LP Performance

Each of the optimizers available in ILOG CPLEX is designed to solve most linear programming problems under its default parameter settings. However, characteristics of your particular problem may make performance tuning advantageous.

As a first step in tuning performance, try the different ILOG CPLEX optimizers, as we suggested in *Choosing an Optimizer for Your LP Problem* on page 124. The following sections describe other features of ILOG CPLEX to consider in tuning the performance of your application:

- Preprocessing
- Starting from an Advanced Basis
- Simplex Parameters

Preprocessing

Under default settings, ILOG CPLEX preprocesses problems by simplifying constraints, reducing problem size, and eliminating redundancy. Its presolver tries to reduce the size of a problem by making inferences about the nature of any optimal solution to the problem. Its aggregator tries to eliminate variables and rows through substitution. For most models,

preprocessing is beneficial to the total solution speed, and ILOG CPLEX reports the model's solution in terms of the user's original formulation, making the exact nature of any reductions immaterial.

In cases where your model may be primal infeasible or unbounded (dual infeasible), it may be desirable to control the kinds of presolve reductions which ILOG CPLEX makes, in order to make your analysis of the outcome of optimization more certain. These reductions can be divided into two types: primal reductions and dual reductions. A reduction is primal if it does not depend on the objective function. A reduction is dual if it does not depend on the right hand side. By default, presolve performs both kinds of reductions.

Under the default, if the presolved model is infeasible, we know only that the original model is either infeasible or unbounded. But if presolve has performed only primal reductions and the presolved model is infeasible, then we have full assurance that the original model is also infeasible. Similarly if presolve has performed only dual reductions and the presolved model is unbounded, then the original model is verified as unbounded.

To control the types of reductions performed by the presolver, set the `Reduce` parameter to one of the following values:

0 = no primal and dual reductions
1 = only primal reductions
2 = only dual reductions
3 = both primal and dual reductions (default)

A useful preprocessing feature for performance tuning, one that is not always activated by default, can be to convert the problem to its dual formulation. The nature of the dual formulation is rooted in linear programming theory, beyond the scope of this manual, but for the purposes of this preprocessing feature it is sufficient to think of the roles of the rows and columns of the model's constraint matrix as being switched. Thus the feature is especially applicable to models that have many more rows than columns. When you solve with the barrier optimizer, the default preprocessor settings may result in the dual formulation automatically being formed if it appears to be advantageous. You can still direct the preprocessor to form the dual with other optimizers by setting the `PreDual` parameter to `1`. Conversely, to entirely inhibit the dual formulation for the barrier optimizer, you can set the `PreDual` parameter to `-1`. The default, automatic, setting is `0`.

It is worth emphasizing, to those familiar with linear programming theory, that the decision to solve the dual formulation of your model, via this preprocessing parameter, is entirely separate from the choice between using the dual simplex method or the primal simplex method to perform the optimization. Although these features have theoretical foundations in common, it is valid to consider, for example, solving the dual formulation of your model with the dual simplex method; this would not simply result in the same computational path as solving the primal formulation with the primal simplex method. In the case already mentioned of a model with many more rows than columns, either simplex variant may perform much better when solving the dual formulation, due to the smaller basis matrix that is maintained.

The ILOG CPLEX preprocessor offers a dependency checker which strengthens problem reduction by detecting redundant constraints. Such reductions are usually most effective with the barrier optimizer. To turn on the dependency checker to strengthen reduction, set the parameter `DepInd` to `1` instead of its default setting of `0`.

To reduce memory usage, presolve may compress the arrays used for storage of the original model. This can make more memory available for the use of the optimizer that the user has called. Under default settings, ILOG CPLEX will only perform this compression when using the out-of-core variant of the barrier optimizer (discussed in *Solving LP Problems with the Barrier Optimizer* on page 156). You can explicitly turn this feature on or off by setting the presolve compression parameter `PreCompress` to `-1` for off, or `1` for on; the default of `0` specifies the automatic setting.

In rare instances, a user may wish to specify the number of analysis passes that the presolver or the aggregator makes through the problem. The parameters `PrePass` and `AggInd`, respectively, control these two preprocessing features; the default, `automatic`, setting of `-1` lets ILOG CPLEX determine the number of passes to make, while a setting of `0` directs ILOG CPLEX to not use that preprocessing feature, and a positive integer limits the number of passes to that value. Under the automatic setting, ILOG CPLEX applies the aggregator just once when it is solving an LP model; for some problems, it may be worthwhile to increase the `AggInd` setting. The behavior under the `PrePass` default is less easy to predict, but if the output log indicates it is performing excessive analysis you may wish to try a limit of five passes or some other modest value.

Another parameter, which affects only the aggregator, is `AggFill`. Occasionally the substitutions made by the aggregator will increase matrix density and thus make each iteration too expensive to be advantageous. In such cases, try lowering `AggFill` from its default value of `10`. ILOG CPLEX may make fewer substitutions as a consequence, and the resulting problem will be less dense.

Finally, if for some reason you wish to turn ILOG CPLEX preprocessing entirely off, set the parameter `PreInd` to `0`.

Starting from an Advanced Basis

As another performance improvement, consider starting from an advanced basis. (The primal simplex, dual simplex, and network optimizers can start optimizing from an advanced basis if one is available; the barrier optimizer does not start from an advanced basis.) If you can start from an advanced basis, then ILOG CPLEX may iterate significantly fewer times, particularly if your current problem is similar to a problem that you have solved previously. Even when problems are different, starting from an advanced basis may still help performance. For example, if your current problem is composed of several smaller problems, an optimal basis from one of the component problems may significantly speed up solution of the larger problem.

Note that if you are solving a sequence of LP models all within one run, by entering a model, solving it, modifying the model, and solving it again, under default settings the advanced basis will be used for the last of these steps automatically.

In other cases, you can communicate the final basis from one run to the start of the next by first saving the basis to a file before the end of the first run.

To save the basis of the optimized problem to a file:

- When using the Component Libraries to optimize your problem, save by means of the method `cplex.exportModel()` or the routine `CPXwriteprob()`.
- In the Interactive Optimizer, use the `write` command with the file type `bas`.

To read an advanced basis from a saved file into the Interactive Optimizer, follow these steps:

1. Assure that the advanced start parameter is set to its default value of `yes`: `set advance y`.
2. Read the file with the `read` command, and specify the file type as `bas`.

Similarly, when using the Component Libraries, set the parameter `IloCplex::AdvInd` or `CPX_PARAM_ADVIND`, to `1` and call the method `IloCplex.importModel()` or the routine `CPXreadcopybase()`.

Simplex Parameters

After you have chosen the right optimizer and, if appropriate, you have started from an advanced basis, you may want to experiment with different parameter settings to improve performance. This section describes parameters that are most likely to affect performance of the simplex linear optimizers. (The barrier optimizer is different enough from the simplex optimizers that it is discussed elsewhere in this chapter, in *Solving LP Problems with the Barrier Optimizer* on page 156). The simplex tuning suggestions are arranged according to the following groupings:

- Pricing Algorithm and Gradient Parameters
- Scaling
- Crash
- Memory Management and Problem Growth

Pricing Algorithm and Gradient Parameters

The parameters in Table 5.3 determine the pricing algorithms that ILOG CPLEX uses. Consequently, these are the algorithmic parameters most likely to affect simplex linear programming performance. The default setting of these gradient parameters choose the pricing algorithms that are best for most problems. When you are selecting alternate pricing algorithms, look at these values as guides:

- overall solution time;
- number of Phase I iterations (that is, iterations before ILOG CPLEX arrives at an initial feasible solution);
- total number of iterations.

ILOG CPLEX records those values in the log file as it works. (By default, ILOG CPLEX creates the log file in the directory where it is executing, and it names the log file `cplex.log`. *Managing Log Files: the Log File Parameter* on page 295 tells you how to rename and relocate this log file.)

If the number of iterations required to solve your problem is approximately the same as the number of rows, then you are doing well. If the number of iterations is three times greater than the number of rows (or more), then it may very well be possible to improve performance by changing the parameter that determines the pricing algorithm, `DPriInd` for the dual simplex optimizer or `PPriInd` for the primal simplex optimizer.

The symbolic names for the acceptable values for these parameters are given in Table 5.3 and Table 5.4 below. The default value in both cases is `0`.

Table 5.3 *DPriInd Parameter Values for Dual Simplex Pricing Algorithm*

	Description	Concert	Callable Library
`0`	determined automatically	`DPriIndAuto`	`CPX_DPRIIND_AUTO`
`1`	standard dual pricing	`DPriIndFull`	`CPX_DPRIIND_FULL`
`2`	steepest-edge pricing	`DPriIndSteep`	`CPX_DPRIIND_STEEP`
`3`	steepest-edge in slack space	`DPriIndFullSteep`	`CPX_DPRIIND_FULLSTEEP`
`4`	steepest-edge, unit initial norms	`DPriIndSteepQStart`	`CPX_DPRIIND_STEEPQSTART`

Table 5.4 *PPriInd Parameter Values for Primal Simplex Pricing Algorithm*

	Description	Concert	Callable Library
`-1`	reduced-cost pricing	`PPriIndPartial`	`CPX_PPRIIND_PARTIAL`
`0`	hybrid reduced-cost and devex	`PPriIndAuto`	`CPX_PPRIIND_AUTO`
`1`	devex pricing	`PPriIndDevex`	`CPX_PPRIIND_DEVEX`
`2`	steepest-edge pricing	`PPriIndSteep`	`CPX_PPRIIND_STEEP`
`3`	steepest-edge, slack initial norms	`PPriIndSteepQStart`	`CPX_PPRIIND_STEEPQSTART`
`4`	full pricing	`PriIndFull`	`CPX_PPRIIND_FULL`

For the dual simplex pricing parameter, the default value selects steepest-edge pricing. That is, the default (0 or `CPX_DPRIIND_AUTO`) automatically selects `2` or `CPX_DPRIIND_STEEP`.

For the primal simplex pricing parameter, *reduced-cost pricing* (`-1`) is less computationally expensive, so you may prefer it for small or relatively easy problems. Try reduced-cost

pricing, and watch for faster solution times. Also if your problem is dense (say, 20-30 nonzeros per column), reduced-cost pricing may be advantageous.

In contrast, if you have a more difficult problem taking many iterations to complete Phase I and arrive at an initial solution, then you should consider *devex pricing* (`1`). Devex pricing benefits more from ILOG CPLEX linear algebra enhancements than does partial pricing, so it may be an attractive alternative to partial pricing in some problems. Do *not* use devex pricing, however, if your problem has many columns and relatively few rows. In such a case, the number of calculations required per iteration will usually be disadvantageous.

If you observe that devex pricing helps, then you might also consider steepest-edge pricing (`2`). Steepest-edge pricing is computationally more expensive than reduced-cost pricing, but it may produce the best results on difficult problems. One way of reducing the computational intensity of steepest-edge pricing is to choose steepest-edge pricing with initial slack norms (`3`).

Scaling

Poorly conditioned problems (that is, problems in which even minor changes in data result in major changes in solutions) may benefit from an alternative *scaling* method. Scaling attempts to rectify poorly conditioned problems by multiplying rows or columns by constants without changing the fundamental sense of the problem. If you observe that your problem has difficulty staying feasible during its solution, then you should consider an alternative scaling method.

Scaling is determined by the parameter `ScaInd`, and may be set to any of the following values:

Table 5.5 *ScaIndParameter Values for Scaling Methods*

ScaInd Value	Meaning
`-1`	no scaling
`0`	equilibration scaling (default)
`1`	aggressive scaling

Refactorization Frequency

ILOG CPLEX dynamically determines the frequency at which the basis of a problem is refactorized in order to maximize iteration speed. On very large problems, ILOG CPLEX refactorizes the basis matrix infrequently. Very rarely should you consider increasing the number of iterations between refactorizing. The refactorization interval is controlled by the `ReInv` parameter. The default value of `0` means ILOG CPLEX will decide dynamically; any positive integer specifies the user's chosen factorization frequency.

Crash

It is possible to control the way ILOG CPLEX builds an initial (crash) basis through the `CraInd` parameter.

In the dual simplex optimizer, the `CraInd` setting determines whether ILOG CPLEX aggressively uses primal variables instead of slack variables while it still tries to preserve as much dual feasibility as possible. If its value is `1` (one), it indicates the default starting basis; if its value is `0` (zero) or `-1`, it indicates an aggressive starting basis. These values are summarized in Table 5.6.

Table 5.6 *CraInd Parameter Values for the Dual Simplex Optimizer*

CraInd Value	Meaning for Dual Simplex Optimizer
`1`	Use default starting basis
`0`	Use an aggressive starting basis
`-1`	Use an aggressive starting basis

In the primal simplex optimizer, the `CraInd` setting determines how ILOG CPLEX uses the coefficients of the objective function to select the starting basis. If its value is `1` (one), ILOG CPLEX uses the coefficients to guide its selection; if its value is `0` (zero), ILOG CPLEX ignores the coefficients; if its value is `-1`, ILOG CPLEX does the opposite of what the coefficients normally suggest. These values are summarized in Table 5.7.

Table 5.7 *CraInd Parameter Values for the Primal Simplex Optimizer*

CraInd Value	Meaning for Primal Simplex Optimizer
`1`	Use coefficients of objective function to select basis
`0`	Ignore coefficients of objective function
`-1`	Select basis contrary to one indicated by coefficients of objective function

Memory Management and Problem Growth

ILOG CPLEX automatically handles memory allocations to accommodate the changing size of a problem object as you modify it. And it manages (using a cache) most changes to prevent inefficiency when the changes will require memory re-allocations. If an application builds a large problem in small increments, you may be able to improve performance by increasing the growth parameters, `RowGrowth`, `ColGrowth`, and `NzGrowth`. In particular, if you know reasonably accurate upper bounds on the number of rows, columns, and nonzeros, and you are building a large problem in very small pieces with many calls to the problem modification routines, then setting the growth parameters to the known upper bounds will

make ILOG CPLEX perform the fewest allocations and may thus improve performance of the problem modification routines. However, overly generous upper bounds may result in excessive memory use.

Diagnosing Performance Problems

While some linear programming models offer opportunities for performance tuning, others, unfortunately, entail outright performance problems that require diagnosis and correction. This section indicates how to diagnose and correct such performance problems as lack of memory or numerical difficulties.

Lack of Memory

To sustain computational speed, ILOG CPLEX should use only available physical memory, rather than virtual memory or paged memory. Even if your problem data fit in memory, ILOG CPLEX will need still more memory to optimize the problem. When ILOG CPLEX recognizes that only limited memory is available, it automatically makes algorithmic adjustments to compensate. These adjustments almost always reduce optimization speed. If you detect when these automatic adjustments occur, then you can determine when you need to add additional memory to your computer to sustain computational speed for your particular problem. The following sections offer guidance for you to detect these automatic adjustments.

Warning Messages

In certain cases, ILOG CPLEX issues a warning message when it cannot perform an operation, but it continues to work on the problem. Other ILOG CPLEX messages indicate that ILOG CPLEX is compressing the problem to conserve memory. These warnings mean that ILOG CPLEX finds insufficient memory available, so it is following an alternate—less desirable—path to a solution. If you provide more memory, ILOG CPLEX will return to the best path toward a solution.

Paging Virtual Memory

If you observe paging of memory to disk, then your application is incurring a performance penalty. If you increase available memory in such a case, performance will speed up dramatically.

Refactorization Frequency and Memory Requirements

The ILOG CPLEX primal and dual simplex optimizers refactorize the problem basis at a rate determined by the `ReInv` parameter.

The longer ILOG CPLEX works between refactorization, the greater the amount of memory it needs for each iteration. Consequently, one way of conserving memory is to decrease the interval between refactorizations. In fact, if little memory is available to it, ILOG CPLEX

will automatically decrease the refactorization interval in order to use less memory at each iteration.

Since refactorization is an expensive operation, decreasing the refactorization interval will generally slow performance. You can tell whether performance is being degraded in this way by checking the iteration log file.

In an extreme case, lack of memory may force ILOG CPLEX to refactorize at every iteration, and the impact on performance will be dramatic. If you provide more memory in such a situation, the benefit will be tremendous.

Preprocessing and Memory Requirements

By default, ILOG CPLEX automatically preprocesses your problem before optimizing, and this preprocessing requires memory. If memory is extremely tight, consider turning off preprocessing, by setting the `PreInd` parameter to `0`. But doing this foregoes the potential performance benefit of preprocessing, and should be considered only as a last resort. Alternatively, consider turning on presolve compression using the `PreCompress` parameter.

Numerical Difficulties

ILOG CPLEX is designed to handle the numerical difficulties of linear programming automatically. In this context, numerical difficulties mean such phenomena as:

- repeated occurrence of singularities;
- little or no progress toward reaching the optimal objective function value;
- little or no progress in scaled infeasibility;
- repeated problem perturbations; and
- repeated occurrences of the solution becoming infeasible.

While ILOG CPLEX will usually achieve an optimal solution in spite of these difficulties, you can help it do so more efficiently. This section describes situations in which you can help.

Some problems will not be solvable even after you take the measures we suggest. Problems can be so poorly conditioned that their optimal solutions are beyond the numerical precision of your computer.

Numerically Sensitive Data

There is no absolute link between the form of data in a model and the numerical difficulty the problem poses. Nevertheless, certain choices in how you present the data to ILOG CPLEX can have an adverse effect.

Placing large upper bounds (say, in the neighborhood of $1e^9$ to $1e^{12}$) on individual variables can cause difficulty during Presolve. If you intend for such large bounds to mean "no bound is really in effect" it is better to simply not include such bounds in the first place.

Large coefficients anywhere in the model can likewise cause trouble at various points in the solution process. Even if the coefficients are of more modest size, a wide variation (say, six or more orders of magnitude) in coefficients found in the objective function or right hand side, or in any given row or column of the matrix, can cause difficulty either in Presolve when it makes substitutions, or in the optimizer routines, particularly the barrier optimizer, as convergence is approached.

A related source of difficulty is the form of rounding when fractions are represented as decimals; expressing $1/3$ as .33333333 on a computer that in principle computes values to about 16 digits can introduce an apparent "exact" value that will be treated as given but may not represent what you intend. This difficulty is compounded if similar or related values are represented a little differently elsewhere in the model.

For example, consider the constraint $1/3$ `x1` + $2/3$ `x2 = 1`. Under perfect arithmetic, it is equivalent to `x1 + 2 x2 = 3`. However, if you express the fractional version with decimal data values, some truncation is unavoidable. If you happen to include the truncated decimal version of the constraint in the same model with some differently-truncated version or even the exact-integer data version, an unexpected result could easily occur. Consider the following problem formulation:

```
Maximize
 obj: x1 + x2
Subject To
 c1: 0.333333 x1 + 0.666667 x2  = 1
 c2: x1 + 2 x2  = 3
End
```

Under default numerical tolerances, this will deliver an optimal solution of `x1=1.0` and `x2=1.0`, giving an objective function value of `2.0`. Now, see what happens when using slightly more accurate data (in terms of the fractional values that are clearly intended to be expressed):

```
Maximize
 obj: x1 + x2
Subject To
 c1: 0.333333333 x1 + 0.666666667 x2  = 1
 c2: x1 + 2 x2  = 3
End
```

The solution to this problem has `x1=3.0` and `x2=0.0`, giving an optimal objective function value of `3.0`, a result qualitatively different from that of the first model. Since this latter result is the same as would be obtained by removing constraint `c1` from the model entirely, this is a more satisfactory result. Moreover, the numerical stability of the optimal basis (as indicated by the condition number, discussed in the next section), is vastly improved.

The result of the extra precision of the input data is a model that is less likely to be sensitive to rounding error or other effects of solving problems on finite-precision computers, or in extreme cases will be more likely to produce an answer in keeping with the intended model. The first example, above, is an instance where the data truncation has fundamentally distorted the problem being solved. Even if the exact-integer data version of the constraint is not present with the decimal version, the truncated decimal version no longer exactly represents the intended meaning and, in conjunction with other constraints in your model, could give unintended answers that are "accurate" in the context of the specific data being fed to the optimizer.

Be particularly wary of data in your model that has been computed (within your program, or transmitted to your program from another via an input file) using single-precision (32 bit) arithmetic. For example in C this would come from using type *float* instead of *double*. Such data will be accurate to only about 8 decimal digits, so that for instance if you print the data you might see values like `0.3333333432674408` instead of `0.3333333333333333`. ILOG CPLEX uses double-precision (64 bit) arithmetic in its computations, and truncated single-precision data carries the risk that it will convey a different meaning than the user intends.

The underlying principle behind all the cautions in this section is that information contained in the data needs to reflect actual meaning or the optimizer may reach unstable solutions or encounter algorithmic difficulties.

Measuring Problem Sensitivity with Basis Condition Number

Ill-conditioned matrices are sensitive to minute changes in problem data. That is, in such problems, small changes in data can lead to very large changes in the reported problem solution. ILOG CPLEX provides a *basis condition number* to measure the sensitivity of a linear system to the problem data. You might also think of the basis condition number as the number of places in precision that can be lost.

For example, if the basis condition number at optimality is $1e^{13}$, then a change in a single matrix coefficient in the thirteenth place may dramatically alter the solution. Furthermore, since many computers provide about 16 places of accuracy in double precision, only three accurate places are left in such a solution. Even if an answer is obtained, perhaps only the first three significant digits are reliable.

Because of this effective loss of precision for matrices with high basis condition numbers, ILOG CPLEX may be unable to select an optimal basis. In other words, a high basis condition number can make it impossible to find a solution.

- In the Interactive Optimizer, use the command `display solution kappa` in order to see the basis condition number of a resident basis matrix.
- In the Concert Technology Library, use the method
  `cplex.getQuality(IloCplex::Kappa)` (C++)
  `cplex.getQuality(IloCplex.QualityType.Kappa)` (Java).

- In the Callable Library, use the routine `CPXgetdblquality()` to access the condition number in the double-precision variable `dvalue`, like this:

```
status = CPXgetdblquality(env, lp, &dvalue, CPX_KAPPA);
```

Repeated Singularities

Whenever ILOG CPLEX encounters a singularity, it removes a column from the current basis and proceeds with its work. ILOG CPLEX reports such actions to the log file (by default) and to the screen (if you are working in the Interactive Optimizer or if the message-to-screen indicator `CPX_PARAM_SCRIND` is set to `1` (one)). Once it finds an optimal solution under these conditions, ILOG CPLEX will then re-include the discarded column to be sure that no better solution is available. If no better objective value can be obtained, then the problem has been solved. Otherwise, ILOG CPLEX continues its work until it reaches optimality.

Occasionally, new singularities occur, or the same singularities recur. ILOG CPLEX observes a limit on the number of singularities it tolerates. The parameter `SingLim` specifies this limit. By default, the limit is ten. After encountering this many singularities, ILOG CPLEX will save in memory the best factorizable basis found so far and stop its solution process. You may want to save this basis to a file for later use.

To save the best factorizable basis found so far in the Interactive Optimizer, use the `write` command with the file type `bas`. When using the Component Libraries, use the method `cplex.exportModel()` or the routine `CPXwriteprob()`.

If ILOG CPLEX encounters repeated singularities in your problem, you may want to try alternative scaling on the problem (rather than simply increasing ILOG CPLEX tolerance for singularities). *Scaling* on page 132 explains how to try alternative scaling.

If alternate scaling does not help, another tactic to try is to increase the Markowitz tolerance. The Markowitz tolerance controls the kinds of pivots permitted. If you set it near its maximum value of `0.99999`, it may make iterations slower but more numerically stable. *Inability to Stay Feasible* on page 139 shows how to change the Markowitz tolerance.

If none of these ideas help, you may need to alter the model of your problem. Consider removing the offending variables manually from your model, and review the model to find other ways to represent the functions of those variables.

Stalling Due to Degeneracy

Highly degenerate linear programs tend to stall optimization progress in the primal and dual simplex optimizers. When stalling occurs with the primal simplex optimizer, ILOG CPLEX automatically perturbs the *variable bounds*; when stalling occurs with the dual simplex optimizer, ILOG CPLEX perturbs the *objective function*.

In either case, perturbation creates a different but closely related problem. Once ILOG CPLEX has solved the perturbed problem, it removes the perturbation by resetting problem data to their original values.

If ILOG CPLEX automatically perturbs your problem early in the solution process, you should consider starting the solution process yourself with a perturbation. (Starting in this way will save the time that would be wasted if you first allowed optimization to stall and then let ILOG CPLEX perturb the problem automatically.)

To start perturbation yourself, set the parameter `PerInd` to `1` instead of its default value of `0`. The perturbation constant, `EpPer`, is usually appropriate at its default value of $1e^{-6}$, but can be set to any value $1e^{-8}$ or larger.

If you observe that your problem has been perturbed more than once, then consider whether the simplex perturbation-limit parameter is too large. The perturbation-limit parameter, `PerLim`, determines the number of iterations ILOG CPLEX tries before it assumes the problem has stalled. At its default value, `0` (zero), ILOG CPLEX determines internally how many iterations to perform before declaring a stall. If you set this parameter to some other nonnegative integer, then ILOG CPLEX uses that limit to determine when a problem has stalled. If you reduce the perturbation-limit parameter, ILOG CPLEX may be able to solve the problem more smoothly by perturbing sooner.

Inability to Stay Feasible

If a problem repeatedly becomes infeasible in Phase II (that is, after ILOG CPLEX has achieved a feasible solution), then numerical difficulties may be occurring. It may help to increase the Markowitz tolerance in such a case. By default, the value of the parameter `EpMrk` is `0.01`, and suitable values range from `0.0001` to `0.99999`.

Sometimes slow progress in Phase I (the period when ILOG CPLEX calculates the first feasible solution) is due to similar numerical difficulties, less obvious because feasibility is not gained and lost. In the progress reported in the log file, an increase in the printed sum of infeasibilities may be a symptom of this case. If so, it may be worthwhile to set a higher Markowitz tolerance, just as in the more obvious case of numerical difficulties in Phase II.

Diagnosing LP Infeasibility

ILOG CPLEX reports statistics about any problem that it optimizes. For infeasible solutions, it reports values that you can analyze to determine where your problem formulation proved infeasible. In certain situations, you can then alter your problem formulation or change ILOG CPLEX parameters to achieve a satisfactory solution.

- When the ILOG CPLEX *primal* simplex optimizer terminates with an infeasible basic solution, it calculates dual variables and reduced costs relative to the Phase I objective; that is, relative to the infeasibility function. The Phase I objective function depends on the current basis. Consequently, if you use the primal simplex optimizer with various parameter settings, an infeasible problem will produce different objective values and different solutions.

- In the case of the dual simplex optimizer, termination with a status of infeasibility occurs only during Phase II. Therefore all solution values are relative to the problem's natural (primal) formulation, including the values related to the objective function such as the dual variables and reduced costs. As with the primal simplex optimizer, the basis at which the determination of infeasibility is made may not be unique.

The Effect of Preprocessing on Feasibility

ILOG CPLEX preprocessing may declare a model infeasible before the selected optimization algorithm begins. This saves considerable execution time in most cases. It is important, when this is the outcome, to understand that there are two classes of reductions performed by the preprocessor.

Reductions that are independent of the objective function are called primal reductions; those that are independent of the right-hand side are called dual reductions. Preprocessing operates on the assumption that the model being solved is expected by the user to be feasible and that a finite optimal solution exists. If this assumption is false, then the model is either infeasible or no bounded optimal solutions exist, i.e. unbounded. Since primal reductions are independent of the objective function, they cannot detect unboundedness, they can only detect infeasibility. Similarly, dual reductions can only detect unboundedness.

Thus, to aid analysis of an infeasible or unbounded declaration by the preprocessor, a parameter is provided that the user can set, so that the optimization can be rerun to assure that the results reported by the preprocessor can be interpreted. If a model is declared by the preprocessor to be infeasible or unbounded and the user believes that it might be infeasible, the parameter `Reduce` can be set to `1` by the user, and the preprocessor will only perform primal reductions. If the preprocessor still finds inconsistency in the model, it will be declared by the preprocessor to be infeasible, instead of infeasible or unbounded. Similarly, setting the parameter to `2` means that if the preprocessor detects unboundedness in the model, it will be declared unambiguously to be unbounded.

These parameters are intended for diagnostic use, as turning off reductions will usually have a negative impact on performance of the optimization algorithms in the normal (feasible and bounded) case.

Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities

By default, ILOG CPLEX *scales* a problem before attempting to solve it. After it finds an optimal solution, it then checks for any violations of optimality or feasibility in the original, *unscaled* problem. If there is a violation of reduced cost (indicating nonoptimality) or of a bound (indicating infeasibility), ILOG CPLEX reports both the maximum scaled and unscaled feasibility violations.

Unscaled infeasibilities are rare, but they may occur when a problem is ill-conditioned. For example, a problem containing a row in which the coefficients have vastly different magnitude is ill-conditioned in this sense and may result in unscaled infeasibilities.

It may be possible to produce a better solution anyway in spite of unscaled infeasibilities, or it may be necessary for you to revise the coefficients. To determine which way to go, we recommend these steps in such a case:

1. Use the command `display solution quality` in the Interactive Optimizer to locate the infeasibilities.
2. Examine the coefficient matrix for poorly scaled rows and columns.
3. Evaluate whether you can change unnecessarily large or small coefficients.
4. Consider alternate scalings.

You may also be able to re-optimize the problem successfully after you reduce optimality tolerance, as explained in *Maximum Reduced-Cost Infeasibility* on page 142, or after you reduce feasibility tolerance, as explained in *Maximum Bound Infeasibility: Identifying Largest Bound Violation* on page 142. When you change these tolerances, ILOG CPLEX may produce a better solution to your problem, but lowering these tolerances sometimes produces erratic behavior or an unstable optimal basis.

Check the basis condition number, as explained in *Measuring Problem Sensitivity with Basis Condition Number* on page 137. If the condition number is fairly low (for example, as little as $1e^5$ or less), then you can be confident about the solution. If the condition number is high, or if reducing tolerance does not help, then you must revise the model because the current model may be too ill-conditioned to produce a numerically reliable result.

Interpreting Solution Statistics

By default, individual infeasibilities are written to a log file but not displayed on the screen. To display the infeasibilities on your screen, use the command `set output logonly y cplex.log`.

Regardless of whether a solution is infeasible or optimal, the command `display solution quality` in the Interactive Optimizer displays the bound and reduced-cost infeasibilities for both the scaled and unscaled problem. In fact, it displays the following summary statistics for both the scaled and unscaled problem:

- maximum bound infeasibility, that is, the largest bound violation;
- maximum reduced-cost infeasibility;
- maximum row residual;
- maximum dual residual;
- maximum absolute value of a variable, a slack variable, a dual variable, and a reduced cost.

The following sections discuss these summary statistics in greater detail.

Maximum Bound Infeasibility: Identifying Largest Bound Violation

The maximum bound infeasibility identifies the largest bound violation. This information may help you determine the cause of infeasibility in your problem. If the largest bound violation exceeds the feasibility tolerance of your problem by only a small amount, then you may be able to get a feasible solution to the problem by increasing the feasibility tolerance parameter, `EpRHS`. Its range is between $1e^{-9}$ and 0.1. Its default value is $1e^{-6}$.

Maximum Reduced-Cost Infeasibility

The maximum reduced-cost infeasibility identifies a value for the optimality tolerance that would cause ILOG CPLEX to perform additional iterations. It refers to the infeasibility in the dual slack associated with reduced costs. Whether ILOG CPLEX terminated with an optimal or infeasible solution, if the maximum reduced-cost infeasibility is only slightly smaller in absolute value than the optimality tolerance, then solving the problem with a smaller optimality tolerance may result in an improvement in the objective function.

To change the optimality tolerance, set the parameter `EpOpt`.

Maximum Row Residual

The maximum row residual identifies the maximum constraint violation. ILOG CPLEX Simplex Optimizers control these residuals only indirectly by applying numerically sound methods to solve the given linear system. When ILOG CPLEX terminates with an infeasible solution, all infeasibilities will appear as bound violations on structural or slack variables, not constraint violations. The maximum row residual may help you determine whether a model of your problem is poorly scaled, or whether the final basis (whether it is optimal or infeasible) is ill-conditioned.

Maximum Dual Residual

The maximum dual residual indicates whether the current optimality tolerance is set appropriately. If the maximum dual residual exceeds the optimality tolerance, ILOG CPLEX may stall before it reaches an optimal solution. In particular, if ILOG CPLEX stalls during Phase I after almost reducing the sum of infeasibilities to `0` (zero), then you may be able to find a feasible solution if you increase the optimality tolerance parameter, `EpOpt`.

Maximum Absolute Values: Detecting Ill-Conditioned Problems

When you are trying to determine whether your problem is ill-conditioned, you need to consider the following maximum absolute values, all available in the infeasibility analysis that ILOG CPLEX provides you:

- variables;
- slack variables;
- dual variables;
- reduced costs (that is, dual slack variables).

When using the Component Libraries, use the method `cplex.getQuality()` or the routine `CPXgetdblquality()` to access the information provided by the command `display solution quality` in the Interactive Optimizer.

If you determine from this analysis that your model is indeed ill-conditioned, then you need to reformulate it. *Coping with an Ill-Conditioned Problem or Handling Unscaled Infeasibilities* on page 140 outlines steps to follow in this situation.

Finding a Set of Irreducibly Inconsistent Constraints

If ILOG CPLEX reports that your problem is infeasible, then you should invoke the ILOG CPLEX infeasibility finder to save time and effort in the development process. This diagnostic tool computes a set of infeasible constraints and column bounds that would be feasible if one of them (a constraint or variable) were removed. Such a set is known as an *irreducibly inconsistent set* (IIS).

To work, the infeasibility finder must have a problem that satisfies two conditions:

- the problem has been optimized using the primal simplex optimizer or barrier with primal crossover, and
- the problem has terminated with an infeasible basic solution to the primal problem.

When the ILOG CPLEX presolver detects infeasibility during preprocessing, no optimization has yet taken place. Furthermore, since the presolver may perform many passes on a problem, the reason that it identifies a row as infeasible may not be obvious. To run the infeasibility finder and to see solution statistics in such a case, you should first turn off ILOG CPLEX preprocessing before you optimize, as explained in *Preprocessing* on page 127, before you invoke the infeasibility finder.

The barrier optimizer may be called with primal crossover, dual crossover, or no crossover. Only the primal crossover options will allow you to subsequently run the infeasibility finder.

The parameter `IISInd` controls the method used for determining the IIS. Its default value of `0` (zero) invokes an algorithm that requires minimal computation time but may generate a large set of inconsistent constraints.

Infeasibility Finder in the Interactive Optimizer

To invoke the infeasibility finder and to display part of its findings in the Interactive Optimizer, use the command `display iis`. By default, ILOG CPLEX records all its findings in a log file. To send these findings to your screen as well, use the command `set output logonly y cplex.log`.

You can also write an IIS file from the Interactive Optimizer and then examine it with your preferred text editor to see all the constraints and bounds in the irreducibly inconsistent set.

For an example of how to use the infeasibility finder and how to interpret its results, see *Example: Output from the Infeasibility Finder in the Interactive Optimizer* on page 144.

Infeasibility Finder in the Component Libraries

Its alternative value of `1` (one) may take longer but generates a minimal set of irreducibly inconsistent constraints. Use the method `cplex.out()` or the routines `CPXdisplayiis()` or `CPXiiswrite()` to output the results for review.

Correcting Multiple Infeasibilities

The infeasibility finder will find only one irreducibly inconsistent set (IIS), though a given problem may contain many independent IISs. Consequently, even after you detect and correct one such IIS in your problem, it may still remain infeasible. In such a case, you need to run the infeasibility finder more than once to detect those multiple causes of infeasibility in your problem.

Example: Output from the Infeasibility Finder in the Interactive Optimizer

After you have optimized a problem and ILOG CPLEX has terminated with a primal infeasible basic solution, then you can invoke the ILOG CPLEX infeasibility finder on this optimized problem and its infeasible basic solution. The ILOG CPLEX infeasibility finder will compute an IIS of constraints and column bounds from your problem and record this IIS in a log file along with other useful information to help you locate the source of infeasibility and aid in revising or reformulating your model.

If you want ILOG CPLEX to display this additional information on your screen in the Interactive Optimizer, use the command `set output logonly yes`. After that command, invoke the infeasibility finder with the command `display iis. ILOG CPLEX` will respond like this:

```
Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Performing column sensitivity filter

Number of rows in the iis: 3
Number of columns in the iis: 3
Names of rows in the iis:
NODE5    (fixed)
D7       (lower bound)
D8       (lower bound)
Names of columns in the iis:
T25      (upper bound)
T35      (upper bound)
T47      (upper bound)
Iis Computation Time =    0.01 sec.
```

As you can see, ILOG CPLEX states how many rows and columns comprise the IIS. It also displays the row and column names, and it identifies the bound causing the infeasibility. In this example, all the columns in the IIS are limited by their upper bound. If you remove any of the upper bounds on those columns, then the IIS becomes feasible. The bound information about rows is really needed only for *ranged* rows. In the case of ranged rows, the bound indicates whether the row lies at the lower or upper end of the range of right-hand side values. For other kinds of rows, there is only one possible right-hand side value at

which the row can lie. Greater-than constraints must lie at their lower bound. Less-than constraints must lie at their upper bound. Equality constraints are fixed.

Example: Writing an IIS-Type File

After you have invoked the infeasibility finder with the `display iis` command, if you want additional information to determine the cause of infeasibility, use the `write` command and the file type `iis` to generate a ILOG CPLEX LP format file containing each individual constraint and bound in the IIS. You can then use the `xecute` command to call an ordinary text editor during your ILOG CPLEX session to examine the contents of that IIS file. It will look something like this example:

```
CPLEX> write infeas.iis

Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Performing column sensitivity filter
Irreducibly inconsistent set written to file 'infeas.iis'.

CPLEX> xecute edit infeas.iis

Minimize
subject to
\Rows in the iis:
 NODE5: T25 + T35 - T57 - T58  = 0
 D7:    T47 + T57 >= 20
 D8:    T58 >= 30
\Columns in the iis:
Bounds
 T25 <= 10
 T35 <= 10
 T47 <= 2
\Non-iis columns intersecting iis rows:
 T57 Free
 T58 Free
```

In this example, you see that the bounds on *T25* and *T35* combine with the row *NODE5* to imply that $T57 + T58 \leq 20$. However, row *D7* and the bound on *T47* imply that $T57 \geq 18$. Since row *D8* requires $T58 \geq 30$, we see that $T57 + T58 \geq 48$, so the constraints and bounds are infeasible. Notice that every constraint and bound contributes to this infeasibility, according to the definition of an IIS. There are, in consequence, many different ways to modify such a problem to make it feasible. The "right" change will depend on your knowledge of the problem.

When ILOG CPLEX records the constraints and bounds of an IIS, it also lists as *free* all columns that intersect one or more rows in the IIS but do not have bounds in the IIS. This portion of the file ensures that when you read the file into ILOG CPLEX, the resulting problem satisfies the definition of an IIS. After you read in such a file, you can perform additional problem analysis within your ILOG CPLEX session.

Example: Interpreting a Cumulative Constraint

In the example that we have been looking at, there were sufficiently few row and column bounds in the IIS for us to see the cause of infeasibility at a glance. In contrast, other IIS files may contain so many rows and columns that it becomes difficult to see the cause of infeasibility. When an IIS contains many equality constraints and only a few bounds, for example, this phenomenon commonly occurs. In such a situation, the equality constraints transfer the infeasibility from one part of the model to another until eventually no more transfers can occur. Consequently, such an IIS file will also contain a cumulative constraint consisting of the sum of all the equality rows. This cumulative constraint can direct you toward the cause of infeasibility, as the following sample IIS illustrates:

```
Minimize
subject to
\Rows in the iis:
 2:   - x24 + x97 + x98 - x99 - x104  = -7758
 3:   - x97 + x99 + x100 - x114 - x115  = 0
 4:   - x98 + x104  = 0
 10:  - x105 + x107 + x108 - x109  = -151
 11:  - x108 + x109 + x110 - x111  = -642
 12:  - x101 - x102 - x110 + x111 + x112 + x113 - x117  = -2517
 13:  - x112 + x117 + x118 - x119  = -186
 14:  - x118 + x119 + x120 + x121 - x122 - x123  = -271
 15:  - x120 + x122  = -130
 16:  - x121 + x123 + x124 - x125  = -716
 17:  - x124 + x125 + x126 - x127  = -2627
 18:  - x126 + x127 + x128 - x129  = -1077
 19:  - x128 + x129 + x130 - x131  = -593
 249: - x100 + x101 + x103  = 0
 251: - x113 + x114 + x116  = 0
\Sum of equality rows in iis:
\  - x24 - x102 + x103 - x105 + x107 - x115 + x116 + x130 - x131 = -16668
\Columns in the iis:
Bounds
 x24 <= 14434
 x102 = 0
 x103 = 0
 x105 = 0
 x107 = 0
 x115 = 0
 x116 = 0
 x130 = 0
 x131 = 0
\Non-iis columns intersecting iis rows:
 x97 Free
 x98 Free
 x99 Free
.
.
.
End
```

Since there are 15 rows in this IIS file, the cause of infeasibility is not immediately obvious. However, if we look at the sum of the bounds on the columns, we see that $x24 - x102 + x103 - x105 + x107 - x115 + x116 + x130 - x131 \geq -1443$, so it is impossible

to satisfy the sum of equality rows. Therefore, to correct the infeasibility, we must alter one or more of the bounds, or we must increase one or more of the right-hand side values.

Example: Setting a Time Limit on the Infeasibility Finder

The ILOG CPLEX infeasibility finder will stop when its total runtime exceeds the limit set by the command `set timelimit`. The infeasibility finder works by removing constraints and bounds that cannot be involved in the IIS, so it can provide partial information about an IIS when it reaches its time limit. The collection of constraints and bounds it offers then do not strictly satisfy the definition of an IIS. However, the collection does contain a true IIS within it, and frequently it provides enough information for you to diagnose the cause of infeasibility in your problem. When it reaches the time limit, ILOG CPLEX output indicates that it has found only a partial IIS. The following example illustrates this idea. In it, we set a time limit and then invoke the feasibility finder.

```
CPLEX> set timelimit 2
CPLEX> display iis

Starting Infeasibility Finder Algorithm...
Performing row sensitivity filter
Infeasibility Finder Algorithm exceeded time limit.
Partial infeasibility output available.

Number of rows in the (partial) iis: 101
Number of columns in the (partial) iis: 99
```

Tactics for Interpreting IIS Output

The size of the IIS reported by ILOG CPLEX depends on many factors in the model. If an IIS contains hundreds of rows and columns, you may find it hard to determine the cause of the infeasibility. Fortunately, there are tactics to help you interpret IIS output:

- Consider selecting an alternative IIS algorithm. The default algorithm emphasizes computation speed, and it may give rise to a relatively large IIS. If so, try setting the `IISInd` parameter to `1` (one) to invoke the alternative algorithm, and then run the infeasibility finder again. Normally, the resulting IIS is smaller because the alternative algorithm emphasizes finding a minimal IIS at the expense of computation speed.
- If the problem contains equality constraints, examine the cumulative constraint consisting of the sum of the equality rows. As we illustrated in one of the examples, the cumulative constraint can simplify your interpretation of the IIS output. More generally, if you take other linear combinations of rows in the IIS, that may also help. For example, if you add an equality row to an inequality row, the result may yield a simpler inequality row.
- Try preprocessing with the ILOG CPLEX presolver and aggregator. The presolver may even detect infeasibility by itself. If not, running the infeasibility finder on the presolved problem may help by reducing the problem size and removing extraneous constraints that do not directly cause the infeasibility but still appear in the IIS. Similarly, running

the infeasibility finder on an aggregated problem may help because the aggregator performs substitutions that may remove extraneous variables that clutter the IIS output. More generally, if you perform substitutions, you may simplify the output so that it can be interpreted more easily.

- Other simplifications of the constraints in the IIS, such as combining variables, multiplying constraints by constants, and rearranging sums, may make it easier to interpret the IIS.

Example: Using a Starting Basis in an LP Problem

We now return to an approach mentioned in the section *Tuning LP Performance*, which is to start with an advanced basis. The following small example demonstrates an approach to setting a starting basis by hand.

Example ilolpex6.cpp

The example, `ilolpex6.cpp`, resembles one you may have studied in the ILOG CPLEX *Getting Started* manual, `ilolpex1.cpp`. This example differs from that one in these ways:

- Arrays are constructed using the `populatebycolumn` method, and thus no command line arguments are needed to select a construction method.
- In the `main` routine, the arrays `cstat` and `rstat` set the status of the initial basis.
- After the problem data has been copied into the problem object, the *basis* is copied by a call to `cplex.setBasisStatuses()`.
- After the problem has been optimized, the iteration count is printed. For the given data and basis, the basis is optimal, so no iterations are required to optimize the problem.

The main program starts by declaring the environment and terminates by calling method `end()` for the environment. The code in between is encapsulated in a try block that catches all Concert Technology exceptions and prints them to the C++ error stream `cerr`. All other exceptions are caught as well, and a simple error message is issued. Next the model object and the `cplex` object are constructed. The function `populatebycolumn()` builds the problem object and, as we noted earlier, `cplex.setBasisStatuses()` copies the advanced starting basis.

The call to `cplex.solve()` optimizes the problem, and the subsequent print of the iteration count demonstrates that the advanced basis took effect. In fact, this basis is immediately determined to be the optimal one, resulting in zero iterations being performed, in contrast to the behavior seen in the example program `ilolpex1.cpp` where the same model is solved without the use of an advanced basis.

Complete Program

The complete program, `ilolpex6.cpp`, appears here and online in the standard distribution

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilolpex6.cpp
// Version 8.1
// ------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// ilolpex6.cpp - Illustrates that optimal basis can be copied and
//                used to start an optimization.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN


static void
   populatebycolumn (IloModel model, IloNumVarArray var, IloRangeArray rng);

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env, "example");

      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      populatebycolumn (model, var, rng);

      IloCplex cplex(model);

      IloCplex::BasisStatusArray cstat(env), rstat(env);
      cstat.add(IloCplex::AtUpper);
      cstat.add(IloCplex::Basic);
      cstat.add(IloCplex::Basic);
      rstat.add(IloCplex::AtLower);
      rstat.add(IloCplex::AtLower);
      cplex.setBasisStatuses(cstat, var, rstat, rng);
      cplex.solve();

      cplex.out() << "Solution status = " << cplex.getStatus() << endl;
      cplex.out() << "Solution value  = " << cplex.getObjValue() << endl;
      cplex.out() << "Iteration count = " << cplex.getNiterations() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
```

```
      cplex.getSlacks(vals, rng);
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, rng);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("lpex6.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


static void
populatebycolumn (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 35.0, 40.0));
   x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
   x.add(obj(3.0) + c[0]( 1.0) + c[1]( 1.0));

   model.add(obj);
   model.add(c);
}  // END populatebycolumn
```

Example lpex6.c

The example, `lpex6.c`, resembles one you may have studied in the ILOG CPLEX *Getting Started* manual, `lpex1.c`. This example differs from that one in these ways:

- In the `main` routine, the arrays `cstat` and `rstat` set the status of the initial basis.
- After the problem data has been copied into the problem object, the *basis* is copied by a call to `CPXcopybase()`.
- After the problem has been optimized, the iteration count is printed. For the given data and basis, the basis is optimal, so no iterations are required to optimize the problem.

The application begins with declarations of arrays to store the solution of the problem. Then, before it calls any other ILOG CPLEX routine, the application invokes the Callable Library routine `CPXopenCPLEX()` to initialize the ILOG CPLEX environment. Once the environment has been initialized, the application calls other ILOG CPLEX Callable Library routines, such as `CPXsetintparam()` with the argument `CPX_PARAM_SCRIND` to direct output to the screen and most importantly, `CPXcreateprob()` to create the problem object. The routine `populatebycolumn()` builds the problem object, and as we noted earlier, `CPXcopybase()` copies the advanced starting basis.

Before the application ends, it calls `CPXfreeprob()` to free space allocated to the problem object and `CPXcloseCPLEX()` to free the environment.

Complete Program

The complete program, `lpex6.c`, appears here and online in the standard distribution

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex6.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex6.c - Illustrates that optimal basis can be copied and
             used to start an optimization. */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);



/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6
```

```
int
main (void)
{
   char     probname[16];  /* Problem name is max 16 characters */
   int      cstat[NUMCOLS];
   int      rstat[NUMROWS];

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               "Failure to turn on screen indicator, error %d.\n", status);
      goto TERMINATE;
```

```
}

/* Create the problem. */

strcpy (probname, "example");
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout. */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data. */

status = populatebycolumn (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to populate problem data.\n");
   goto TERMINATE;
}

/* We assume we know the optimal basis.  Variables 1 and 2 are basic,
   while variable 0 is at its upper bound */

cstat[0] = CPX_AT_UPPER;
cstat[1] = CPX_BASIC;
cstat[2] = CPX_BASIC;

/* The row statuses are all nonbasic for this problem */

rstat[0] = CPX_AT_LOWER;
rstat[1] = CPX_AT_LOWER;

/* Now copy the basis */

status = CPXcopybase (env, lp, cstat, rstat);
if ( status ) {
   fprintf (stderr, "Failed to copy the basis.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXlpopt (env, lp);
```

```
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }


   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n", objval);
   printf ("Iteration count = %d\n\n", CPXgetitcnt (env, lp));

   /* The size of the problem should be obtained by asking CPLEX what
      the actual size is, rather than using sizes from when the problem
      was built.  cur_numrows and cur_numcols store the current number
      of rows and columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "lpex6.sav", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }
```

```
   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
```

```
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */
```

Solving LP Problems with the Barrier Optimizer

This section tells you more about solving linear programming problems using the ILOG CPLEX *Barrier Optimizer*. It includes sections on:

- Identifying LPs for Barrier Optimization
- Interpreting the Barrier Log File
- Understanding Solution Quality from the Barrier LP Optimizer
- Overcoming Numerical Difficulties
- Diagnosing Barrier Optimizer Infeasibility

Identifying LPs for Barrier Optimization

The ILOG CPLEX Barrier Optimizer is well suited to large, sparse problems. An alternative to the simplex optimizers, it exploits a primal-dual logarithmic barrier algorithm to generate a sequence of strictly positive primal and dual solutions to a problem. Like the simplex optimizers, it is not really necessary to understand the internal workings of barrier in order to obtain its performance benefits. However, for the interested reader, we first pause to give an outline of how it works.

ILOG CPLEX finds the primal solutions, conventionally denoted *(x, s)*, from the primal formulation:

Minimize $\boldsymbol{c}^T x$

subject to $\boldsymbol{A}x = \boldsymbol{b}$

with these bounds $x + s = u$ and $x \geq l$

where $\boldsymbol{A}$ is the constraint matrix, including slack and surplus variables; u is the upper and l the lower bounds on the variables.

Simultaneously, ILOG CPLEX automatically finds the dual solutions, conventionally denoted *(y, z, w)* from the corresponding dual formulation:

Maximize $b^T y - u^T w + l^T z$

subject to $\boldsymbol{A}^T y - w + z = \boldsymbol{c}$

with these bounds $w \geq 0$ and $z \geq 0$

All possible solutions maintain strictly positive primal solutions $(x - l, s)$ and strictly positive reduced costs (z, w) so that the value 0 (zero) forms a *barrier* for primal and dual variables within the algorithm.

ILOG CPLEX measures progress by considering the primal feasibility, dual feasibility, and duality gap at each iteration. To measure feasibility, ILOG CPLEX considers the accuracy with which the primal constraints ($\boldsymbol{A}x = b$, $x + s = u$) and dual constraints ($\boldsymbol{A}^T y + z - w = \boldsymbol{c}$) are satisfied. The optimizer stops when it finds feasible primal and dual solutions that are *complementary.* A complementary solution is one where the sums of the products $(x_j - l_j)z_j$ and $(u_j - x_j)z_j$ are within some tolerance of 0 (zero). Since each $(x_j - l_j)$, $(u_j - x_j)$, and z_j is

strictly positive, the sum can be near zero only if each of the individual products is near zero. The sum of these products is known as the *complementarity* of the problem.

On each iteration of the barrier optimizer, ILOG CPLEX computes a matrix based on AA^T and then computes a *Cholesky factor* of it. This factored matrix has the same number of nonzeros on each iteration. The number of nonzeros in this matrix is influenced by the barrier *ordering* parameter.

The ILOG CPLEX Barrier Optimizer is appropriate and often advantageous for large problems, for example, those with more than 1000 rows or columns. It is effective on problems with staircase structures or banded structures in the constraint matrix. It is also effective on problems with a small number of nonzeros per column.

Its performance is most dependent on these characteristics:

- the number of floating-point operations required to compute the Cholesky factor;
- the presence of dense columns, that is, columns with a relatively high number of nonzero entries.

To decide whether to use the barrier optimizer on a given problem, you should look at both these characteristics. (We explain how to check those characteristics later in this chapter in *"Cholesky Factor in the Log File"* on page 162, and *"Nonzeros in Lower Triangle of AAT in the Log File"* on page 162).

Barrier Simplex Crossover

Since many users prefer basic solutions because they can be used to restart simplex optimization, the ILOG CPLEX Barrier Optimizer includes *basis crossover algorithms*. By default, the barrier optimizer automatically invokes a primal crossover when the barrier algorithm terminates (unless termination occurs abnormally because of insufficient memory or numerical difficulties). Optionally, you can also execute barrier optimization with a dual crossover or with no crossover at all. The section *"Controlling Crossover"* on page 160 explains how to control crossover in the Interactive Optimizer.

Differences between Barrier and Simplex Optimizers

The barrier optimizer and the simplex optimizers (primal and dual) are fundamentally different approaches to solving linear programming problems. The key differences between them have these implications:

- Simplex and barrier optimizers differ with respect to the *nature of solutions*. Barrier solutions tend to be midface solutions. In cases where multiple optima exist, barrier solutions tend to place the variables at values between their bounds, whereas in basic solutions from a simplex technique, the values of the variables are more likely to be at either their upper or their lower bound. While objective values will be the same, the nature of the solutions can be very different.

  The fact that barrier without crossover does not produce a basic solution has other consequences. Without a basis, you will not be able to optimize the same or similar

problems repeatedly using advanced start information. You will also not be able to obtain range information for performing sensitivity analysis.

- Simplex and barrier optimizers have different *numerical properties*, sensitivity, and behavior. For example, the barrier optimizer is sensitive to the presence of unbounded optimal faces, whereas the simplex optimizers are not. As a result, problems that are numerically difficult for one method may be easier to solve by the other.
- Simplex and barrier optimizers have different *memory requirements*. Depending on the size of the Cholesky factor, the barrier optimizer can require significantly more memory than the simplex optimizers.
- Simplex and barrier optimizers work well on different *types of problems*. The barrier optimizer works well on problems where the AA^T remains sparse. Also, highly degenerate problems that pose difficulties for the primal or dual simplex optimizers may be solved quickly by the barrier optimizer. In contrast, the simplex optimizers will probably perform better on problems where the AA^T and the resulting Cholesky factor are relatively dense, though it is sometimes difficult to predict from the dimensions of the model when this will be the case.

Using the Barrier Optimizer

We have described how the ILOG CPLEX Barrier Optimizer finds primal and dual solutions from the primal and dual formulations of a model (see the section *Identifying LPs for Barrier Optimization* on page 157) , but you do not have to reformulate the problem yourself. The ILOG CPLEX Barrier Optimizer automatically creates the primal and dual formulations of the problem for you after you enter or read in the problem.

Specify that you want to use the barrier optimizer by setting the parameter `LPMethod` to the following value:

`IloCplex::Barrier`	(Concert for C++ users)
`IloCplex.Algorithm.Barrier`	(Concert for Java users)
`CPX_ALG_BARRIER`	(Callable Library)
`4`	(Interactive Optimizer)

And then you call the solution routine just as for any other ILOG CPLEX optimizer.

`cplex.solve()`	(Concert, for C++ or Java users)
`CPXlpopt()`	(Callable Library)
`optimize`	(Interactive Optimizer)

Special Options

In addition to the parameters available for other ILOG CPLEX LP optimizers, there are also parameters to control the ILOG CPLEX Barrier Optimizer. In the Interactive Optimizer, to

see a list of the parameters specific to the ILOG CPLEX Barrier Optimizer, use the command `set barrier`.

Controlling Crossover

The nature of the crossover step that follows barrier is controlled by the parameter `BarCrossAlg`. Under the default `Automatic` setting, an appropriate crossover step will be invoked. Possible settings for the parameter are:

Table 5.8 *BarCrossAlg Parameter Values*

BarCrossAlg Values	Meaning
-1	no crossover
0	automatic (default)
1	primal crossover
2	dual crossover

Using VEC File Format

When you use the ILOG CPLEX Barrier Optimizer with no crossover, you can save the primal and dual variable values and their associated reduced cost and dual values in a VEC-format file. You can then read that VEC file into ILOG CPLEX before you initiate a crossover at a later time. After you read a VEC file into ILOG CPLEX, all three optimizers—primal simplex, dual simplex, and barrier simplex—automatically invoke crossover.

Even if you have set the advanced basis indicator to `no` (meaning that you do not intend to start from an advanced basis), ILOG CPLEX automatically resets the indicator to `yes` when it reads a VEC file. If you turn off the advanced basis indicator after reading a VEC file, then the simplex optimizers will honor this setting and will not initiate crossover.

Interpreting the Barrier Log File

Like the ILOG CPLEX simplex optimizers, the barrier optimizer records information about its progress in a log file as it works. Some users find it helpful to keep a new log file for each session. By default, ILOG CPLEX records information in a file named `cplex.log`. In the:

- Interactive Optimizer, use the command `set logfile filename` to change the name of the log file.
- Callable Library, use the routine `CPXsetlogfile()` with arguments to indicate the log file.

You can control the level of information that ILOG CPLEX records about barrier optimization by setting the `BarDisplay` parameter. Its allowed values and their meanings are as follows:

Table 5.9 *BarDisplay Parameter Values*

BarDisplay Values	Meaning
0	no display
1	display normal information (default)
2	display detailed (diagnostic) output

Here is an example of a log file for a barrier optimization (without crossover):

```
Tried aggregator 1 time.
LP Presolve eliminated 9 rows and 11 columns.
Aggregator did 6 substitutions.
Reduced LP has 12 rows, 15 columns, and 38 nonzeros.
Presolve time =    0.00 sec.
Number of nonzeros in lower triangle of A*A' = 26
Using Approximate Minimum Degree ordering
Total time for automatic ordering = 0.00 sec.
Summary statistics for Cholesky factor:
  Rows in Factor            = 12
  Integer space required    = 12
  Total non-zeros in factor = 78
  Total FP ops to factor    = 650
 Itn      Primal Obj        Dual Obj  Prim Inf Upper Inf  Dual Inf
   0  -1.3177911e+01  -1.2600000e+03  6.55e+02  0.00e+00  3.92e+01
   1  -4.8683118e+01  -5.4058675e+02  3.91e+01  0.00e+00  1.18e+01
   2  -1.6008142e+02  -3.5969226e+02  1.35e-13  7.11e-15  5.81e+00
   3  -3.5186681e+02  -6.1738305e+02  1.59e-10  1.78e-15  5.16e-01
   4  -4.5808732e+02  -4.7450513e+02  5.08e-12  1.95e-14  4.62e-02
   5  -4.6435693e+02  -4.6531819e+02  1.66e-12  1.27e-14  1.59e-03
   6  -4.6473085e+02  -4.6476678e+02  5.53e-11  2.17e-14  2.43e-15
   7  -4.6475237e+02  -4.6475361e+02  5.59e-13  2.99e-14  2.19e-15
   8  -4.6475312e+02  -4.6475316e+02  1.73e-13  1.55e-14  1.17e-15
   9  -4.6475314e+02  -4.6475314e+02  1.45e-13  2.81e-14  2.17e-15

Barrier - Optimal:  Objective =   -4.6475314194e+02
Solution time =    0.01 sec.  Iterations = 9
```

Preprocessing in the Log File

The opening lines of that log file record information about *preprocessing* by the ILOG CPLEX presolver and aggregator. After those preprocessing statistics, the next line records the number of *nonzeros in the lower triangle* of a particular matrix, $\boldsymbol{AA^T}$, denoted `A*A'` in the log file.

Nonzeros in Lower Triangle of AA^T in the Log File

The number of nonzeros in the lower triangle of $\boldsymbol{AA^T}$ gives an early indication of how long each barrier iteration will take. The larger this number, the more time each barrier iteration requires. If this number is close to 50% of the square of the number of rows, then the problem may contain dense columns that are not being detected. In that case, examine the histogram of column counts; then consider setting the barrier column-nonzeros parameter to a value that enables ILOG CPLEX to treat more columns as being dense.

Ordering-Algorithm Time in the Log File

After the number of nonzeros in the lower triangle of $\boldsymbol{AA^T}$, ILOG CPLEX records the time required by the ordering algorithm. (The ILOG CPLEX Barrier Optimizer offers you a choice of four ordering algorithms, explained in *Choosing an Ordering Algorithm* on page 167.) This section in the log file indicates which ordering algorithm the default `Automatic` setting chose.

Cholesky Factor in the Log File

After the time required by the ordering algorithm, ILOG CPLEX records information about the *Cholesky factor*. ILOG CPLEX computes this matrix on each iteration. The *number of rows* in the Cholesky factor represents the number after preprocessing. The next line of information about the Cholesky factor—integer space required—indicates the amount of memory needed to store the *sparsity pattern* of the factored matrix. If this number is low, then the factor can be computed more quickly than when the number is high.

Information about the Cholesky factor continues with the number of nonzeros in the factored matrix. The difference between this number and the number of nonzeros in $\boldsymbol{AA^T}$ indicates the *fill-level* of the Cholesky factor. If the fill-level is large, consider an alternate ordering algorithm.

The final line of information indicates how many floating-point operations are required to compute the Cholesky factor. This number is the best predictor of the time that will be required to perform each iteration of the barrier method.

Iteration Progress in the Log File

After the information about the Cholesky factor, the log file records progress at each iteration. It records both *primal* and *dual objectives* (as `Primal Obj` and `Dual Obj`) per iteration.

It also records absolute infeasibilities per iteration. Internally, the ILOG CPLEX Barrier Optimizer treats inequality constraints as equality constraints with added slack and surplus variables. Consequently, primal constraints in a problem are written as $\boldsymbol{A}x = b$ and $x + s = u$,

and the dual constraints are written as $\boldsymbol{A}^T y + z - w = c$. As a result, in the log file, the infeasibilities represent *norms*, as summarized in Table 5.10.

Table 5.10 *Infeasibilities and Norms in the Log File of a Barrier Optimization*

Infeasibility	In log file	Norm
primal	`Prim Inf`	|b - **A**x|
upper	`Upper Inf`	|u - (x + s)|
dual	`Dual Inf`	|c - y**A** - z + w|

If solution values are large in absolute value, then the infeasibilities may appear inordinately large because they are recorded in the log file in absolute terms. The optimizer uses *relative* infeasibilities as termination criteria.

Infeasibility Ratio in the Log File

If you are using one of the barrier infeasibility algorithms available in the ILOG CPLEX Barrier Optimizer (that is, if you have set `BarAlg` to either 1 or 2, as discussed later in this chapter), then ILOG CPLEX records an additional column of output titled `Inf Ratio`, the infeasibility ratio. This ratio, always positive, is a measure of progress for that particular algorithm. In a problem with an optimal solution, you will see this ratio increase to a large number. In contrast, in a problem that is infeasible or unbounded, this ratio will decrease to a very small number.

Understanding Solution Quality from the Barrier LP Optimizer

When ILOG CPLEX successfully solves a problem with the ILOG CPLEX Barrier Optimizer, it reports the optimal objective value and solution time in a log file, as it does for other LP optimizers.

Because barrier solutions (prior to crossover) are not basic solutions, certain solution statistics associated with basic solutions are not available for a strictly barrier solution. For example, reduced costs and dual values are available for strictly barrier LP solutions, but range information about them is not.

To help you evaluate the quality of a barrier solution more readily, ILOG CPLEX offers a special display of information about barrier solution quality. To display this information in the Interactive Optimizer, use the command `display solution quality` after optimization. When using the Component Libraries, use the method

`cplex.getQuality()` or use the routines `CPXgetintquality()` for integer information and `CPXgetdblquality()` for double-valued information.

Table 5.11 *Barrier Solution Quality Display*

Item	Meaning
primal objective	primal objective value $c^T x$
dual objective	dual objective value $b^T y - u^T w + l^T z$
duality gap	difference between primal and dual objectives
complementarity	sum of column and row complementarity
column complementarity (total)	sum of $\|(x_j - l_j) \bullet z_j\| + \|(u_j - x_j) \bullet w_j\|$
column complementarity (max)	maximum of $\|(x_j - l_j) \bullet z_j\|$ and $\|(u_j - x_j) \bullet w_j\|$ over all variables
row complementarity (total)	sum of $\|slack_i \bullet y_i\|$
row complementarity (max)	maximum of $\|slack_i \bullet y_i\|$
primal norm |x| (total)	sum of absolute values of all primal variables
primal norm |x| (max)	maximum of absolute values of all primal variables
dual norm |rc| (total)	sum of absolute values of all reduced costs
dual norm |rc| (max)	maximum of absolute values of all reduced costs
primal error (Ax = b) (total, max)	total and maximum error in satisfying primal equality constraints
dual error (A'`pi` + `rc` = c) (total, max)	total and maximum error in satisfying dual equality constraints
primal x bound error (total, max)	total and maximum error in satisfying primal lower and upper bound constraints
primal slack bound error (total, max)	total and maximum violation in slack variables
dual pi bound error (total, max)	total and maximum violation with respect to zero of dual variables on inequality rows
dual rc bound error (total, max)	total and maximum violation with respect to zero of reduced costs

Table 5.11 *Barrier Solution Quality Display (Continued)*

Item	Meaning
primal normalized error (Ax = b) (max)	accuracy of primal constraints
dual normalized error (A'pi + rc = c) (max)	accuracy of dual constraints

Table 5.11 lists the items ILOG CPLEX displays and explains their meaning. In the solution quality display, the term `pi` refers to dual solution values, that is, the y values in the conventional barrier problem-formulation. The term `rc` refers to reduced cost, that is, the difference $z - w$ in the conventional barrier problem-formulation. Other terms are best understood in the context of primal and dual LP formulations.

Normalized errors, for example, represent the accuracy of satisfying the constraints while considering the quantities used to compute $\boldsymbol{A}x$ on each row and $y^T\boldsymbol{A}$ on each column. In the primal case, for each row, we consider the nonzero coefficients and the x_j values used to compute $\boldsymbol{A}x$. If these numbers are large in absolute value, then it is acceptable to have a larger absolute error in the primal constraint.

Similar reasoning applies to the dual constraint.

If ILOG CPLEX returned an optimal solution, but the primal error seems high to you, the primal *normalized* error should be low, since it takes into account the scaling of the problem and solution.

After a simplex optimization—whether primal, dual, or network—or after a crossover, the display command will display information related to the quality of the *simplex* solution.

Tuning Barrier Optimizer Performance

Naturally, the default parameter settings for the ILOG CPLEX Barrier Optimizer work best on most problems. However, you can tune several algorithmic parameters to improve performance or to overcome numerical difficulties. These parameters are described in the sections:

- Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage
- Preprocessing: the Presolver and Aggregator
- Detecting and Eliminating Dense Columns
- Choosing an Ordering Algorithm
- Using a Starting-Point Heuristic

In addition, several parameters set termination criteria. With them, you control when ILOG CPLEX stops optimization.

You can also control convergence tolerance—another factor that influences performance. Convergence tolerance determines how nearly optimal a solution ILOG CPLEX must find: tight convergence tolerance means ILOG CPLEX must keep working until it finds a solution very close to the optimal one; loose tolerance means ILOG CPLEX can return a solution within a greater range of the optimal one and thus stop calculating sooner.

Performance of the ILOG CPLEX Barrier Optimizer is most highly dependent on the number of floating-point operations required to compute the Cholesky factor at each iteration. When you adjust barrier parameters, always check their impact on this number. At default output settings, this number is reported at the beginning of each barrier optimization in the log file, as we explain in *Cholesky Factor in the Log File* on page 162.

Another important performance issue is the presence of *dense columns*. By a dense column, we mean that a given variable appears in a relatively large number of rows. You can check column density as we suggest in *Nonzeros in Lower Triangle of AAT in the Log File* on page 162. We also say more about column density in *Detecting and Eliminating Dense Columns* on page 167.

In adjusting parameters, you may need to experiment to find beneficial settings because the precise effect of parametric changes will depend on the nature of your LP problem as well as your platform (hardware, operating system, compiler, etc.). Once you have found satisfactory parametric settings, keep them in a parameter specification file for re-use, as explained in *Saving a Parameter Specification File* on page 400.

Out-of-Core Barrier: Letting the Optimizer Use Disk for Storage

Under default settings, the ILOG CPLEX Barrier Optimizer will do all of its work using central memory (variously referred to also as RAM, core, or physical memory). For models too large to solve in the central memory on your computer, or in cases where it is simply not desired to use this much memory, it is possible to instruct the barrier optimizer to use disk for part of the working storage it needs, specifically the Cholesky factorization. Since disk is slower than central memory, there may be some lost performance by making this choice on models that could be solved entirely in central memory, but the out-of-core feature in the ILOG CPLEX Barrier Optimizer is designed to make this as efficient as possible. It generally will be far more effective than relying on the operating system's virtual memory (swap space). To activate out-of-core Barrier, set the parameter `BarOOC` to `1` instead of its default value of `0`.

Note that out-of-core barrier uses some working memory to store the portion of the factor on which it is currently performing computation. You can improve performance by allocating more working memory, using the `WorkMem` parameter. More working memory allows the algorithm to transfer less factor data to and from disk. In fact, the factor matrix will not be written to disk at all if its size does not exceed the value of `WorkMem`. The default for this parameter is `128`, meaning 128 megabytes.

When Barrier is being run out-of-core, the location of disk storage is controlled by the Working Directory parameter, `WorkDir`. For example, to use the directory `/tmp/mywork`,

set `WorkDir` to the string '`/tmp/mywork`'. `WorkDir` should be specified as the name of a directory that already exists, and ILOG CPLEX will create its working directory under that.

Preprocessing: the Presolver and Aggregator

For best performance of the ILOG CPLEX Barrier Optimizer, preprocessing should almost always be on. That is, we recommend that you use the default setting where the presolver and aggregator are active. While they may use more memory, they also reduce the problem, and problem reduction is crucial to barrier optimizer performance. In fact, reduction is so important that even when you turn off preprocessing, ILOG CPLEX still applies minimal presolving before barrier optimization.

For problems that contain linearly dependent rows, it is a good idea to turn on the *preprocessing dependency parameter.* (By default, it is off.) This dependency checker may add some preprocessing time, but it can detect and remove linearly dependent rows to improve overall performance.

To turn on dependency checking, set parameter `DepInd` to `1`.

Detecting and Eliminating Dense Columns

Dense columns can significantly degrade barrier optimizer performance. A dense column is one in which a given variable appears in many rows. The interactive optimizer contains a display feature that shows a histogram of the number of nonzeros in the columns of your model, `display problem histogram c`.

In fact, when a few dense columns are present in a problem, it is often effective to reformulate the problem to remove those dense columns from the model.

Otherwise, you can control whether ILOG CPLEX perceives columns as dense by setting the column nonzeros parameter. At its default setting, ILOG CPLEX calculates an appropriate value for this parameter automatically. However, if your problem contains one (or a few) dense columns that remain undetected at the default setting, you can adjust this parameter yourself to help ILOG CPLEX detect it (or them). For example, in a large problem in which one column contains forty entries while the other columns contain less than five entries, you may benefit by setting the column nonzeros parameter to 30. This setting allows ILOG CPLEX to recognize that column as dense and thus invoke techniques to handle it.

To set the dense column threshold, set the parameter `BarColNz` to a positive integer. The default value of `0` means that ILOG CPLEX will determine the threshold.

Choosing an Ordering Algorithm

ILOG CPLEX offers several different algorithms in the ILOG CPLEX Barrier Optimizer for ordering the rows of a matrix:

- automatic, the default, indicated by the value `0`;
- approximate minimum degree (AMD), indicated by the value `1`;

- approximate minimum fill (AMF) indicated by the value 2;
- nested dissection (ND) indicated by the value 3.

The log file, as we explain in *Ordering-Algorithm Time in the Log File* on page 162, records the time spent by the ordering algorithm in a barrier optimization, so you can experiment with different ordering algorithms and compare their performance on your problem.

Automatic ordering, the default option, will usually be the best choice. This option attempts to choose the most effective of the available ordering methods, and it usually results in the best order. It may require more time than the other settings. The ordering time is usually small relative to the total solution time, and a better order can lead to a smaller total solution time. In other words, a change in this parameter is unlikely to improve performance very much.

The AMD algorithm provides good quality order within moderate ordering time. AMF usually provides better order than AMD (usually 5-10% smaller factors) but it requires somewhat more time (10-20% more). ND often produces significantly better order than AMD or AMF. Ten-fold reductions in runtimes of the ILOG CPLEX Barrier Optimizer have been observed with it on some problems. However, ND sometimes produces worse order, and it requires much more time.

To select an ordering algorithm, set the parameter `BarOrder` to a value 0, 1, 2, or 3.

Using a Starting-Point Heuristic

ILOG CPLEX supports several different heuristics to compute the starting point for the ILOG CPLEX Barrier Optimizer. The starting-point heuristic is determined by the `BarStartAlg` parameter, and Table 5.12 summarizes the possible values and their meanings.

Table 5.12 *BarStartAlg Parameter Values for Starting-Point Heuristics*

Value	Heuristic
1	dual is 0 (default)
2	estimate dual
3	average primal estimate, dual 0
4	average primal estimate, estimate dual

For most problems the default works well. However, if you are using the dual preprocessing option (setting the parameter `PreDual` to `1`) then one of the other heuristics for computing a starting point may perform better than the default.

- In the Interactive Optimizer, use the command `set barrier startalg i`, substituting a value for `i`.

- When using the Component Libraries, set the parameter `IloCplex::BarStartAlg` or `CPX_PARAM_BARSTARTALG`.

Overcoming Numerical Difficulties

As we noted in *Differences between Barrier and Simplex Optimizers* on page 159, the algorithms in the barrier optimizer have very different numerical properties from those in the simplex optimizer. While the barrier optimizer is often extremely fast, particularly on very large problems, numerical difficulties occasionally arise with it in certain classes of problems. For that reason, we recommend that you run simplex optimizers in conjunction with the barrier optimizer to verify solutions. At its default settings, the ILOG CPLEX Barrier Optimizer always crosses over after a barrier solution to a simplex optimizer, so this verification occurs automatically.

Difficulties in the Quality of Solution

Understanding Solution Quality from the Barrier LP Optimizer on page 163 lists the items that ILOG CPLEX displays about the quality of a barrier solution. If the ILOG CPLEX Barrier Optimizer terminates its work with a solution that does not meet your quality requirements, you can adjust parameters that influence the quality of a solution. Those adjustments affect the choice of ordering algorithm, the choice of barrier algorithm, the limit on barrier corrections, and the choice of starting-point heuristic—topics introduced in *Tuning Barrier Optimizer Performance* on page 165 and recapitulated here in the following subsections.

Change the Ordering Algorithm

As we explain about tuning performance in *Choosing an Ordering Algorithm* on page 167, you can choose one of several ordering algorithms to use in the ILOG CPLEX Barrier Optimizer. To improve the quality of a solution in some problems, change the ordering algorithm.

Change the Barrier Algorithm

The ILOG CPLEX Barrier Optimizer implements the algorithms listed in Table 5.13. The selection of barrier algorithm is controlled by the `BarAlg` parameter. The default option invokes option `3` for LPs and QPs and option `1` for MIPs where the ILOG CPLEX Barrier Optimizer is used on the subproblems. Naturally, the default is the fastest for most problems, but it may not work well on problems that are primal infeasible or dual infeasible. Options `1` and `2` in the ILOG CPLEX Barrier Optimizer implement a barrier algorithm that also detects infeasibility. (They differ from each other in how they compute a starting point.) Though

they are slower than the default option, in a problem demonstrating numerical difficulties, they may eliminate the numerical difficulties and thus improve the quality of the solution.

Table 5.13 *BarAlg Parameter Values for Barrier Optimizer Algorithm*

BarAlg Value	Meaning
0	default
1	algorithm starts with infeasibility estimate
2	algorithm starts with infeasibility constant
3	standard barrier algorithm

Change the Limit on Barrier Corrections

The default barrier algorithm in the ILOG CPLEX Barrier Optimizer computes an estimate of the maximum number of *centering corrections* that ILOG CPLEX should make on each iteration. You can see this computed value by setting barrier display level two, as explained in *Interpreting the Barrier Log File* on page 160, and checking the value of the parameter to limit corrections. (Its default value is `-1`.) If you see that the current value is `0` (zero), then you should experiment with greater settings. Setting the parameter `BarMaxCor` to a value greater than `0` may improve numerical performance, but there may also be an increase in computation time.

Choose a Different Starting-Point Heuristic

As we explained in *Using a Starting-Point Heuristic* on page 168, the default starting-point heuristic works well for most problems suitable to barrier optimization. But for a model that is exhibiting numerical difficulty it is possible that setting the `BarStartAlg` to select a different starting point will make a difference. However, if you are preprocessing your problem as dual (for example, in the Interactive Optimizer you issued the command `set preprocessing dual`), then a different starting-point heuristic may perform better than the default. To change the starting-point heuristic, see Table 5.12 on page 168.

Difficulties during Optimization

Numerical difficulties can degrade performance of the ILOG CPLEX Barrier Optimizer or even prevent convergence toward a solution. There are several possible sources of numerical difficulties:

- elimination of too many dense columns may cause numerical instability;
- tight convergence tolerance may aggravate small numerical inconsistencies in a problem;
- unbounded optimal faces may remain undetected and thus prevent convergence.

The following subsections offer guidance about overcoming those difficulties.

Numerical Instability Due to Elimination of Too Many Dense Columns

Detecting and Eliminating Dense Columns on page 167 explains how to change parameters to encourage ILOG CPLEX to detect and eliminate as many dense columns as possible. However, in some problems, if ILOG CPLEX removes too many dense columns, it may cause numerical instability.

You can check how many dense columns ILOG CPLEX removes by looking at the preprocessing statistics at the beginning of the log file, as we explained in *Preprocessing in the Log File* on page 161. If you observe that the removal of too many dense columns results in numerical instability in your problem, then increase the column nonzeros parameter, `BarColNz`.

The default value of the column nonzeros parameter is `0` (zero); that value tells ILOG CPLEX to calculate the parameter automatically.

To see the current value of the column nonzeros parameter—either one you have set or one ILOG CPLEX has automatically calculated—you need to look at the level two display, by setting the `BarDisplay` parameter to `2`.

If you determine that the current value of the column nonzeros parameter is inappropriate for your problem and thus tells ILOG CPLEX to remove too many dense columns, then you can *increase* the parameter `BarColNz` to keep the number of dense columns removed *low.*

Small Numerical Inconsistencies and Tight Convergence Tolerance

If your problem contains small numerical inconsistencies, it may be difficult for the ILOG CPLEX Barrier Optimizer to achieve a satisfactory solution at the default setting of the complementarity convergence tolerance. In such a case, you should increase the `BarEpComp` parameter to a value greater than its default, $1e^{-8}$.

Unbounded Variables and Unbounded Optimal Faces

An *unbounded optimal face* occurs in an LP or QP that contains a sequence of optimal solutions, all with the same value for the objective function and unbounded variable values. The ILOG CPLEX Barrier Optimizer will fail to terminate normally if an undetected unbounded optimal face exists.

Normally, the ILOG CPLEX Barrier Optimizer uses its barrier growth parameter, `BarGrowth`, to detect such conditions. If this parameter is increased beyond its default value, the ILOG CPLEX Barrier Optimizer will be less likely to determine that the problem has an unbounded optimal face and more likely to encounter numerical difficulties.

Consequently, you should change the barrier growth parameter *only* if you find that the ILOG CPLEX Barrier Optimizer is terminating its work before it finds the true optimum because it has falsely detected an unbounded face.

The parameter `BarVarUp` determines an upper bound for all variables in the model; the default value of $1e^{+20}$ is effectively treated as infinity.

Difficulties with Unbounded Problems

ILOG CPLEX detects unbounded problems in either of two ways:

- either it finds a solution with small complementarity that is not feasible for either the primal or the dual formulation of the problem;
- or the iterations tend toward infinity with the objective value becoming very large in absolute value.

The ILOG CPLEX Barrier Optimizer stops when the absolute value of either the primal or dual objective exceeds the objective range parameter, `BarObjRng`.

If you increase the value of `BarObjRng`, then the ILOG CPLEX Barrier Optimizer will iterate more times before it decides that the current problem suffers from an unbounded objective value.

If you know that your problem has *large objective values*, consider increasing `BarObjRng`.

Also if you know that your problem has *large objective values*, consider changing the barrier algorithm by resetting the `BarAlg` parameter.

Diagnosing Barrier Optimizer Infeasibility

When the ILOG CPLEX Barrier Optimizer terminates and reports an infeasible solution, all the usual solution information is available. However, the solution values, reduced costs, and dual variables reported then do not correspond to a basis; hence, that information does not have the same meaning as the corresponding output from the ILOG CPLEX simplex optimizers.

Actually, since the ILOG CPLEX Barrier Optimizer works in a single phase, all reduced costs and dual variables are calculated in terms of the *original* objective function.

If the ILOG CPLEX Barrier Optimizer reports to you that a problem is infeasible, but you still need a basic solution for the problem, use the primal simplex optimizer. ILOG CPLEX will then use the solution provided by the barrier optimizer to determine a starting basis for the primal simplex optimizer. When the primal simplex optimizer finishes its work, you will have an infeasible basic solution for further infeasibility analysis.

If the default algorithm in the ILOG CPLEX Barrier Optimizer determines that your problem is primal infeasible or dual infeasible, then try the alternate algorithms in the barrier optimizer. These algorithms, though slower than the default, are better at detecting primal and dual infeasibility.

To select one of the barrier infeasibility algorithms, set the `BarAlg` parameter to either `1` or `2`.

In *Finding a Set of Irreducibly Inconsistent Constraints* on page 143, we explained how to invoke the infeasibility finder on a solution basis found by one of the simplex optimizers. If you are using the *pure* barrier optimizer (that is, with no crossover to a simplex optimizer),

then it will not generate a basis on which you can call the infeasibility finder to analyze your constraints and locate an IIS. Consequently, if you are interested in finding an IIS for your problem, you should invoke the ILOG CPLEX Barrier Optimizer with crossover, as explained in *Controlling Crossover* on page 160.

6

Solving Network-Flow Problems

This chapter tells you more about the ILOG CPLEX Network Optimizer. It includes sections on:

- Choosing an Optimizer: Network Considerations
- Formulating a Network Problem
- Example: Network Optimizer in the Interactive Optimizer
- Example: Using the Network Optimizer with the Callable Library
- Solving Network-Flow Problems as LP Problems
- Example: Network to LP Transformation
- Solving Problems with the Network Optimizer

Choosing an Optimizer: Network Considerations

As we explain in *Using the Callable Library in an Application* on page 85, to exploit ILOG CPLEX in your own application, you must first create a ILOG CPLEX environment, instantiate a problem object, and populate the problem object with data. As your next step, you call a ILOG CPLEX optimizer.

If part of your problem is structured as a network, then you may want to consider calling the ILOG CPLEX Network Optimizer. This optimizer may have a positive impact on performance. There are two alternative ways of calling the network optimizer:

- If your problem is an LP where a large part is a network structure, you may call the network optimizer for the populated LP object.
- If your entire problem consists of a network flow, you should consider creating a *network object* instead of an LP object. Then populate it, and solve it with the network optimizer. This alternative generally yields the best performance because it does not incur the overhead of LP data structures. This option is only available for the C library.

How much performance improvement you observe between using only a simplex optimizer versus using the network optimizer followed by either of the simplex optimizers depends on the number and nature of the other constraints in your problem. On a pure network problem, we have measured performance 100 times faster with the network optimizer. However, if the network component of your problem is small relative to its other parts, then using the solution of the network part of the problem as a starting point for the remainder may or may not improve performance, compared to running the primal or dual simplex optimizer. Only experiments with your own problem can tell.

Formulating a Network Problem

A *network-flow* problem finds the minimal-cost flow through a *network*, where a network consists of a set N of nodes and a set A of arcs connecting the nodes. An arc a in the set A is an ordered pair (i, j) where i and j are nodes in the set N; node i is called the *tail* or the from-node and node j is called the *head* or the to-node of the arc a. Not all the pairs of nodes in a set N are necessarily connected by arcs in the set A. More than one arc may connect a pair of nodes; in other words, $a_1 = (i, j)$ and $a_2 = (i, j)$ may be two different arcs in A, both connecting the nodes i and j in N.

Each arc a may be associated with four values:

- x_a is the flow value, that is, the amount passing through the arc a from its tail (or from-node) to its head (or to-node). The flow values are the modeling variables of a network-flow problem. Negative values are allowed; a negative flow value indicates that there is flow from the head to the tail.
- l_a, the lower bound, determines the minimum flow allowed through the arc a. By default, the lower bound on an arc is 0 (zero).
- u_a, the upper bound, determines the maximum flow allowed through the arc a. By default, the upper bound on an arc is positive infinity.
- c_a, the objective value, determines the contribution to the objective function of one unit of flow through the arc.

Each node n is associated with one value:

- s_n is the supply value at node n.

By convention, a node with strictly positive supply value (that is, $s_n > 0$) is called a *supply* node or a *source*, and a node with strictly negative supply value (that is, $s_n < 0$) is called a *demand* node or a *sink*. A node where $s_n = 0$ is called a *transshipment* node. The sum of all supplies must match the sum of all demands; if not, then the network flow problem is *infeasible*.

T_n is the set of arcs whose tails are node n; H_n is the set of arcs whose heads are node n. The usual form of a network problem looks like this:

Minimize (or maximize) $$\sum_{a \in A} (c_a x_a)$$

subject to $$\sum_{a \in T_n} x_a - \sum_{a \in H_n} x_a = s_n \forall (n \in N)$$

with these bounds $$l_a \leq x_a \leq u_a \forall (a \in A)$$

That is, for each node, the net flow entering and leaving the node must equal its supply value, and all flow values must be within their bounds. The solution of a network-flow problem is an assignment of flow values to arcs (that is, the modeling variables) to satisfy the problem formulation. A flow that satisfies the constraints and bounds is *feasible*.

Example: Network Optimizer in the Interactive Optimizer

This example is based on a network where the aim is to minimize cost and where the flow through the network has both cost and capacity. Figure 6.1 shows you the nodes and arcs of this network. The nodes are labeled by their identifying node number from 1 through 8. The number inside a node indicates its supply value; 0 (zero) is assumed where no number is given. The arcs are labeled 1 through 14. The lower bound `l`, upper bound `u`, and objective value `c` of each arc are displayed in parentheses `(l, u, c)` beside each arc. In this example, node 1 and node 5 are sources, representing a positive net flow, whereas node 4 and node 8 are sinks, representing negative net flow.

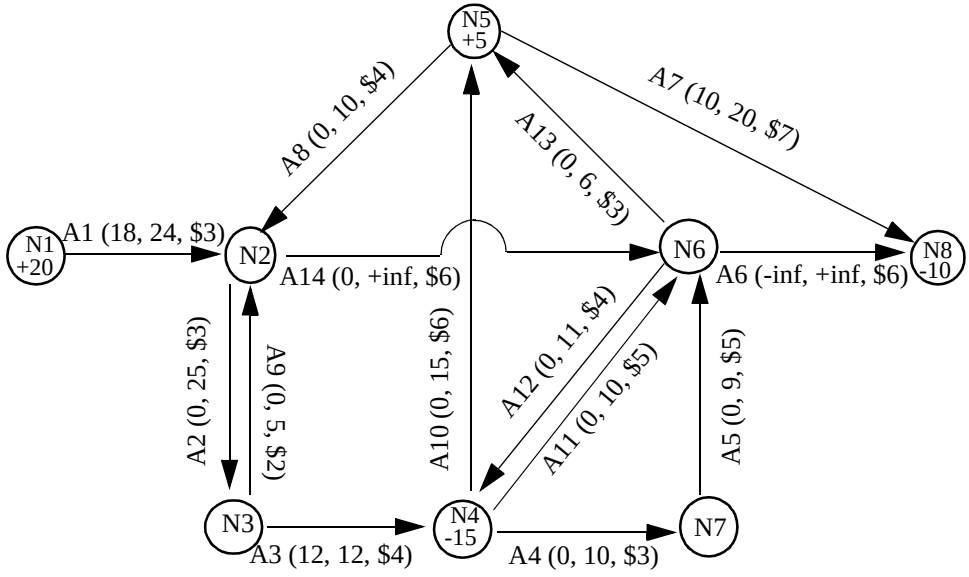

Figure 6.1 *A Directed Network with Arc-Capacity, Flow-Cost, Sinks, and Sources*

In the standard distribution of ILOG CPLEX, the file `nexample.net` contains the formatted problem formulation for this example. You can read it into the Interactive Optimizer with the command `read nexample.net`. After you read the problem into the Interactive Optimizer, you can solve it with the command `netopt` or the command `optimize`.

Understanding the Network Log File

As ILOG CPLEX solves the problem, it produces a log like the following lines:

```
Iteration log . . .
Iteration:     0   Infeasibility     =            48.000000 (5.15396e+13)

Network - Optimal:  Objective =    2.6900000000e+02
Solution time =    0.00 sec.  Iterations = 7 (7)
```

This network log file differs slightly from the log files produced by other ILOG CPLEX optimizers: it contains values enclosed in parentheses that represent modified objective function values.

As long as the network optimizer has not yet found a feasible solution, we say that it is in Phase I. In Phase I, the network optimizer uses modified objective coefficients that penalize infeasibility. At its default settings, the ILOG CPLEX Network Optimizer displays the value of the objective function calculated in terms of these modified objective coefficients in parentheses in the network log file.

You can control the amount of information recorded in the network log file, just as you control the amount of information in other ILOG CPLEX log files. To record no information at all in the log file, use the command `set network display 0`. To display the current objective value in parentheses relative to the actual unmodified objective coefficients, use the command `set network display 1`. To see the display we described earlier in this section, leave the network display parameter at its default value, `2`. (If you have changed the default value, you can reset it with the command `set network display 2`.)

Tuning Performance of the Network Optimizer

The default values of parameters controlling the network optimizer are generally the best choices for effective performance. However, the following sections indicate parameters that you may want to experiment with in your particular problem.

Controlling Tolerance

You control the feasibility tolerance for the network optimizer through the parameter `NetEpRHS`. Likewise, you control the optimality tolerance for the network optimizer through the parameter `NetEpOpt`.

Selecting a Pricing Algorithm for the Network Optimizer

On the rare occasions when the network optimizer seems to take too long to find a solution, you may want to change the pricing algorithm to try to speed up computation. The pricing algorithm for the network optimizer is controlled by parameter `NetPPriInd`. All the choices use variations of partial reduced-cost pricing.

Limiting Iterations in the Network Optimizer

Use the parameter `NetItLim` if you want to limit the number of iterations that the network optimizer performs.

Solving Problems with the Network Optimizer

You instruct ILOG CPLEX to apply the network optimizer for solving the LP at hand by setting the `CPX_PARAM_LPMETHOD` parameter to `CPX_ALG_NET` in the C library, or setting the `RootAlg` parameter to `Network` in the C++ and Java library. When doing so, ILOG CPLEX performs a sequence of steps. It first searches for a part of the LP that conforms to network structure. Such a part is known as an *embedded* network. It then uses the network optimizer to solve that embedded network. Next, it uses the resulting basis to construct a starting basis for the full LP problem. Finally, it solves the LP problem with a simplex optimizer.

You can also use the network optimizer when solving QPs. To do so using the C library, you set the `CPX_PARAM_QPMETHOD` parameter to `CPX_ALG_NET`. For C++ and Java, the `RootAlg` parameter must be set to `Network`. When ILOG CPLEX uses the network

optimizer to solve a QP, it first ignores the quadratic term and uses the network optimizer to solve the resulting LP, as described above. ILOG CPLEX then uses the resulting basis to start a simplex algorithm on the QP model with the original quadratic objective.

Network Extraction

The ILOG CPLEX network extractor searches an LP constraint matrix for a submatrix with the following characteristics:

- the coefficients of the submatrix are all *0* (zero), *1* (one), or *-1 (minus one)*;
- each variable appears in at most two rows with at most one coefficient of *+1* and at most one coefficient of *-1*.

ILOG CPLEX can perform different levels of extraction. The level it performs depends on the `NetFind` parameter.

- When the `NetFind` parameter is set to `1` (one), ILOG CPLEX extracts only the obvious network; it uses no scaling; it scans rows in their natural order; it stops extraction as soon as no more rows can be added to the network found so far.
- When the `NetFind` parameter is set to `2`, the default setting, ILOG CPLEX also uses reflection scaling (that is, it multiplies rows by `-1`) in an attempt to extract a larger network.
- When the `NetFind` parameter is set to `3`, ILOG CPLEX uses general scaling, rescaling both rows and columns, in an attempt to extract a larger network.

In terms of total solution time expended, it may or may not be advantageous to extract the largest possible network. Characteristics of your problem will determine the tradeoff between network size and the number of simplex iterations required to finish solving the model after solving the embedded network.

Even if your problem does not conform precisely to network conventions, the network optimizer may still be advantageous to use. When it is possible to transform the original statement of a linear program into network conventions by these algebraic operations:

- changing the signs of coefficients,
- multiplying constraints by constants,
- rescaling columns,
- adding or eliminating redundant relations,

then ILOG CPLEX will carry out such transformations automatically if you set the `NetFind` parameter appropriately.

Preprocessing and the Network Optimizer

If your LP problem includes network structures, there is a possibility that ILOG CPLEX preprocessing may eliminate those structures from your model. For that reason, you should consider turning off preprocessing before you invoke the network optimizer on a problem.

Example: Using the Network Optimizer with the Callable Library

In the standard distribution of ILOG CPLEX, the file `netex1.c` contains code that creates, solves, and displays the solution of the network-flow problem illustrated in Figure 6.1 on page 178.

Briefly, the `main()` function initializes the ILOG CPLEX environment and creates the problem object; it also calls the optimizer to solve the problem and retrieves the solution.

In detail, `main()` first calls the Callable Library routine `CPXopenCPLEX()`. As we explain in *Initialize the ILOG CPLEX Environment* on page 85, `CPXopenCPLEX()` must always be the first ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. Those routines create the ILOG CPLEX environment and return a pointer (called `env`) to it. This pointer will be passed to every Callable Library routine. If this initialization routine fails, `env` will be `NULL` and the error code indicating the reason for the failure will be written to `status`. That error code can be transformed into a string by the Callable Library routine `CPXgeterrorstring()`.

After `main()` initializes the ILOG CPLEX environment, it uses the Callable Library routine `CPXsetintparam()` to turn on the ILOG CPLEX screen indicator parameter `CPX_PARAM_SCRIND` so that ILOG CPLEX output appears on screen. If this parameter is turned off, ILOG CPLEX does not produce viewable output, neither on screen, nor in a log file. We recommend turning this parameter on when you are debugging your application.

The Callable Library routine `CPXNETcreateprob()` creates an *empty* problem object, that is, a minimum-cost network-flow problem with no arcs and no nodes.

The function `buildNetwork()` populates the problem object; that is, it loads the problem data into the problem object. Pointer variables in the example are initialized as `NULL` so that you can check whether they point to valid data—a good programming practice. The most important calls in this function are to the Callable Library routines, `CPXNETaddnodes()`, which adds nodes with the specified supply values to the network problem, and `CPXNETaddarcs()`, which adds the arcs connecting the nodes with the specified objective values and bounds. In this example, both routines are called with their last argument `NULL` indicating that no names are assigned to the network nodes and arcs. If you want to name arcs and nodes in your problem, pass an array of strings instead.

The function `buildNetwork()` also includes a few routines that are not strictly necessary to this example, but illustrate concepts you may find useful in other applications. To delete a

node and all arcs dependent on that node, it uses the Callable Library routine `CPXNETdelnodes()`. To change the objective sense to minimization, it uses the Callable Library routine `CPXNETchgobjsen()`.

Also `buildNetwork()` sets the row growth and column growth parameters `CPX_PARAM_ROWGROWTH` and `CPX_PARAM_COLGROWTH`. These parameters specify the amount that internal arrays are extended if more nodes or arcs are added than currently fit in allocated memory. If you build up a problem by adding nodes and arcs one by one, and if these parameters are set to a low value, then internal arrays will be frequently reallocated; frequent reallocation may negatively impact performance. Ideally, these parameters are set to the maximum number of nodes and arcs that the problem will ever have. This setting will avoid all reallocation and therefore provide best performance. The parameter `CPX_PARAM_ROWGROWTH` pertains to adding nodes to a network problem (and rows to an LP, QP, or MIP problem) whereas `CPX_PARAM_COLGROWTH` pertains to adding arcs to a network problem (or columns to an LP, QP, or MIP problem).

Let's return to `main()`, where it actually calls the network optimizer with the Callable Library routine, `CPXNETprimopt()`. If `CPXNETprimopt()` returns a nonzero value an error has occurred, otherwise, the optimization was successful. Before retrieving that solution, we allocate arrays to hold it. Then we use `CPXNETsolution()` to copy the solution into those arrays. After we display the solution on screen, we write the network problem into a file, `netex1.net` in the NET file format.

The `TERMINATE:` label is used as a place for the program to exit if any type of error occurs. Therefore, code following this label cleans up: it frees the memory that has been allocated for the solution data; it frees the network object by calling `CPXNETfreeprob()`; and it frees the ILOG CPLEX environment by calling `CPXcloseCPLEX()`. All freeing should be done only if the data is actually available. The Callable Library routine `CPXcloseCPLEX()` should always be the last ILOG CPLEX routine called in a ILOG CPLEX Callable Library application. In other words, all ILOG CPLEX objects that have been allocated should be freed before the call to `CPXcloseCPLEX()`.

Complete Program: netex1.c

The complete program, `netex1.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/netex1.c                                             */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* netex1.c - Entering and optimizing a network problem */

/* Import the CPLEX function declarations and the C library
```

```
   header file stdio.h with the include of cplex.h. */

#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Import the declarations for the string functions */

#include <string.h>



/* Forward declaration for function at end of program */

static int
   buildNetwork  (CPXENVptr env, CPXNETptr net);

static void
   free_and_null (char **ptr);


int
main (void)
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int      narcs;
   int      nnodes;
   int      solstat;
   double   objval;
   double   *x     = NULL;
   double   *pi    = NULL;
   double   *slack = NULL;
   double   *dj    = NULL;

   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   int       status;
   int       i, j;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no
      output, so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }
```

```
/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem. */

net = CPXNETcreateprob (env, &status, "netex1");

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( net == NULL ) {
   fprintf (stderr, "Failed to create network object.\n");
   goto TERMINATE;
}

/* Fill in the data for the problem.  Note that since the space for
   the data already exists in local variables, we pass the arrays
   directly to the routine to fill in the data structures.  */

status = buildNetwork (env, net);

if ( status ) {
   fprintf (stderr, "Failed to build network problem.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXNETprimopt (env, net);
if ( status ) {
   fprintf (stderr, "Failed to optimize network.\n");
   goto TERMINATE;
}

/* get network dimensions */

narcs  = CPXNETgetnumarcs  (env, net);
nnodes = CPXNETgetnumnodes (env, net);

/* allocate memory for solution data */

x     = (double *) malloc (narcs  * sizeof (double));
dj    = (double *) malloc (narcs  * sizeof (double));
pi    = (double *) malloc (nnodes * sizeof (double));
slack = (double *) malloc (nnodes * sizeof (double));
```

```
   if ( x     == NULL ||
        dj    == NULL ||
        pi    == NULL ||
        slack == NULL   ) {
      fprintf (stderr, "Failed to allocate arrays.\n");
      goto TERMINATE;
   }

   status = CPXNETsolution (env, net, &solstat, &objval, x, pi, slack, dj);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }

   /* Write the output to the screen. */

   printf ("\nSolution status = %d\n", solstat);
   printf ("Solution value  = %f\n\n", objval);

   for (i = 0; i < nnodes; i++) {
      printf ("Node %2d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < narcs; j++) {
      printf ("Arc  %2d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXNETwriteprob (env, net, "netex1.net", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write network to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free memory for solution data */

   free_and_null ((char **) &x);
   free_and_null ((char **) &dj);
   free_and_null ((char **) &pi);
   free_and_null ((char **) &slack);

   /* Free up the problem as allocated by CPXNETcreateprob, if necessary */

   if ( net != NULL ) {
      CPXNETfreeprob (env, &net);
      if ( status ) {
         fprintf (stderr, "CPXNETfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
```

```
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
      char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */



static int
buildNetwork (CPXENVptr env, CPXNETptr net)
{
   int status = 0;

   /* definitions to improve readability */

#  define NNODES  8
#  define NARCS  14
#  define inf    CPX_INFBOUND

   /* Define list of supply values for the nodes */

   double supply[NNODES] = {20.0, 0.0, 0.0, -15.0, 5.0, 0.0, 0.0, -10.0};

   /* Define list of tail or from-node indices as well as head or
      to-node indices for the arcs.  Notice that according to C
      standard the first node has index 0. */

   int    tail[NARCS] = {   0,    1,    2,    3,    6,    5,    4,
                            4,    2,    3,    3,    5,    5,    1};
   int    head[NARCS] = {   1,    2,    3,    6,    5,    7,    7,
                            1,    1,    4,    5,    3,    4,    5};

   /* Define list of objective values and lower and upper bound values
      for the arcs */

   double obj [NARCS] = { 3.0,  3.0,  4.0,  3.0,  5.0,  6.0,  7.0,
                          4.0,  2.0,  6.0,  5.0,  4.0,  3.0,  6.0};
   double ub  [NARCS] = {24.0, 25.0, 12.0, 10.0,  9.0,  inf, 20.0,
                         10.0,  5.0, 15.0, 10.0, 11.0,  6.0,  inf};
   double lb  [NARCS] = {18.0,  0.0, 12.0,  0.0,  0.0, -inf,  0.0,
                          0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0};

   /* Delete existing network.  This is not necessary in this
      context since we know we have an empty network object.
      Notice that CPXNETdelnodes deletes all arcs incident to
```

```
      the deleted nodes as well.  Therefore this one function
      call effectively deletes an existing network problem. */

   if ( CPXNETgetnumnodes (env, net) > 0 ) {
      status = CPXNETdelnodes (env, net, 0,
                               CPXNETgetnumnodes (env, net)-1);
      if ( status ) goto TERMINATE;
   }

   /* Set growth rates for rows/nodes and columns/arcs.  This
      is to avoid internal memory reallocations while adding
      nodes and arcs.  Since we are adding all nodes and all
      arcs using only one function call for each it is actually
      unnecessary, but if more function calls are used, finding
      the right settings may improve performance. */

   status = CPXsetintparam (env, CPX_PARAM_ROWGROWTH, NNODES);
   if ( status ) goto TERMINATE;

   status = CPXsetintparam (env, CPX_PARAM_COLGROWTH, NARCS);
   if ( status ) goto TERMINATE;

   /* Set optimization sense */

   status = CPXNETchgobjsen (env, net, CPX_MIN);
   if ( status ) goto TERMINATE;

   /* Add nodes to network along with their supply values,
      but without any names. */

   status = CPXNETaddnodes (env, net, NNODES, supply, NULL);
   if ( status ) goto TERMINATE;

   /* Add arcs to network along with their objective values and
      bounds, but without any names. */

   status = CPXNETaddarcs (env, net, NARCS, tail, head, lb, ub, obj, NULL);
   if ( status ) goto TERMINATE;

TERMINATE:

   return (status);

}  /* END buildnetwork */


static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Solving Network-Flow Problems as LP Problems

A network-flow model is an LP model with special structure. The ILOG CPLEX Network Optimizer is a highly efficient implementation of the primal simplex technique adapted to take advantage of this special structure. In particular, no basis factoring occurs. However, it is possible to solve network models using any of the ILOG CPLEX LP optimizers if first, you convert the network data structures to those of an LP model. To convert the network data structures to LP data structures, in the Interactive Optimizer, use the command `change problem lp`; from the Callable Library, use the routine `CPXcopynettolp()`.

The LP formulation of our example from Figure 6.1 on page 178 looks like this:

Minimize

$$3a_1 + 3a_2 + 4a_3 + 3a_4 + 5a_5 + 6a_6 + 7a_7 + 4a_8 + 2a_9 + 6a_{10} + 5a_{11} + 4a_{12} + 3a_{13} + 6a_{14}$$

subject to

$$\begin{aligned}
a_1 &= 20\\
-a_1 + a_2 - a_8 - a_9 + a_{14} &= 0\\
- a_2 + a_3 + a_9 &= 0\\
- a_3 + a_4 + a_{10} + a11 - a_{12} &= -15\\
a_7 + a_8 - a_{10} - a_{13} &= 5\\
- a_5 + a_6 - a_{11} + a_{12} + a_{13} - a_{14} &= 0\\
- a_4 + a_5 &= 0\\
- a_6 - a_7 &= -10
\end{aligned}$$

with these bounds

$$\begin{aligned}
18 \le a_1 &\le 24\\
0 \le a_2 &\le 25\\
a_3 &= 12\\
0 \le a_4 &\le 10\\
0 \le a_5 &\le 9\\
a_6 &\text{ free}\\
0 \le a_7 &\le 20\\
0 \le a_8 &\le 10
\end{aligned}$$

$$0 \leq a_9 \leq 5$$

$$0 \leq a_{10} \leq 15$$

$$0 \leq a_{11} \leq 10$$

$$0 \leq a_{12} \leq 11$$

$$0 \leq a_{13} \leq 6$$

$$0 \leq a_{14}$$

In that formulation, in each column there is exactly one coefficient equal to *1* (one), exactly one coefficient equal to *-1*, and all other coefficients are *0* (zero).

Since a network-flow problem corresponds in this way to an LP problem, you can indeed solve a network-flow problem by means of a ILOG CPLEX LP optimizer as well. If you read a network-flow problem into the Interactive Optimizer, you can transform it into its LP formulation with the command `change problem lp`. After this change, you can apply any of the LP optimizers to this problem.

When you change a network-flow problem into an LP problem, the basis information that is available in the network-flow problem is passed along to the LP formulation. In fact, if you have already solved the network-flow problem to optimality, then if you call the primal or dual simplex optimizers (for example, with the Interactive Optimizer command `primopt` or `tranopt`), that simplex optimizer will perform no iterations.

Generally, you can also use the same basis from a basis file for both the LP and the network optimizers. However, there is one exception: in order to use an LP basis with the network optimizer, at least one slack variable or one artificial variable needs to be basic. *Starting from an Advanced Basis* on page 129 explains more about this topic in the context of LP optimizers.

If you have already read the LP formulation of a problem into the Interactive Optimizer, you can transform it into a network with the command `change problem network`. Given any LP problem and this command, ILOG CPLEX will try to find the largest network embedded in the LP problem and transform it into a network-flow problem. However, as it does so, it discards all rows and columns that are not part of the embedded network. At the same time, ILOG CPLEX passes along as much basis information as possible to the network optimizer.

Example: Network to LP Transformation

This example shows how to transform a network-flow problem into its corresponding LP formulation. That example also indicates why you might want to make such a change. The example reads a network-flow problem from a file (rather than populating the problem object by adding rows and columns as we did in `netex1.c`). It then attempts to solve the

problem by calling the Callable Library routine `CPXNETprimopt()`. If it determines that the problem is infeasible, it then transforms the problem into its LP formulation so that the infeasibility finder can analyze the problem and possibly indicate the cause of the infeasibility in an irreducibly inconsistent set (IIS). To perform this analysis, the application calls the Callable Library routine `CPXiiswrite()` to write the IIS to the file `netex2.iis`.

Complete Program: netex2.c

The complete program, `netex2.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/netex2.c                                             */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* netex2.c - Reading and optimizing a network problem.
              Transforming to LP to find IIS. */

/* Import the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Import the declarations for the string functions */

include <string.h>



int
main (int argc, char **argv)
{
   /* Declare variables and arrays for retrieving problem data and
      solution information later on. */

   int       status = 0;
   CPXENVptr env = NULL;
   CPXNETptr net = NULL;
   CPXLPptr  lp  = NULL;

   /* Check command line */

   if ( argc != 2 ) {
      fprintf (stderr, "Usage: %s <network file>\n", argv[0]);
      fprintf (stderr, "Exiting ...\n");
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);
```

```
/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no
   output, so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem. */

net = CPXNETcreateprob (env, &status, "netex2");

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( net == NULL ) {
   fprintf (stderr, "Failed to create network object.\n");
   goto TERMINATE;
}

/* Read network problem data from file
   with filename given as command line argument. */

status = CPXNETreadcopyprob (env, net, argv[1]);

if ( status ) {
   fprintf (stderr, "Failed to build network problem.\n");
   goto TERMINATE;
}


/* Optimize the problem */

status = CPXNETprimopt (env, net);
if ( status ) {
   fprintf (stderr, "Failed to optimize network.\n");
   goto TERMINATE;
```

```
   }

   /* Check network solution status */

   if ( CPXNETgetstat (env, net) == CPX_STAT_INFEASIBLE ) {

      /* Create LP object used for invoking infeasibility finder */

      lp = CPXcreateprob (env, &status, "netex2");
      if ( lp == NULL ) {
         fprintf (stderr, "Failed to create LP object.\n");
         goto TERMINATE;
      }

      /* Copy LP representation of network problem to lp object, along
         with the current basis available in the network object. */

      status = CPXcopynettolp (env, lp, net);
      if ( status ) {
         fprintf (stderr, "Failed to copy network as LP.\n");
         goto TERMINATE;
      }

      /* Optimize the LP with primal to create an LP solution.  This
         optimization will start from the basis previously generated by
         CPXNETprimopt() as long as the advance indicator is switched
         on (its default).  */

      status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
      if ( status ) {
         fprintf (stderr,
                  "Failure to set LP method, error %d.\n", status);
         goto TERMINATE;
      }

      status = CPXlpopt (env, lp);
      if ( status ) {
         fprintf (stderr, "Failed to optimize LP.\n");
         goto TERMINATE;
      }

      /* Find IIS and write it to a file */

      status = CPXiiswrite (env, lp, "netex2.iis");
      if ( status ) {
         fprintf (stderr, "Failed to find IIS or write IIS file\n");
         goto TERMINATE;
      }

      printf ("IIS written to file netex2.iis\n");
   }
   else {
      printf ("Network problem not proved to be infeasible\n");
   }


TERMINATE:
```

```
   /* Free up the problem as allocated by CPXNETcreateprob, if necessary */

   if ( net != NULL ) {
      CPXNETfreeprob (env, &net);
      if ( status ) {
         fprintf (stderr, "CPXNETfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */
```

C H A P T E R

7

Solving Quadratic Programming Problems

This chapter tells you about solving *convex quadratic programming* problems (QPs) with ILOG CPLEX.

This chapter contains sections on:

- Identifying Convex Quadratic Programming Problems
- Entering QPs
- Saving QP Problems
- Changing Problem Type in QPs
- Changing Quadratic Terms
- Optimizing QPs
- Example: Creating a QP, Optimizing, Finding a Solution
- Example: Reading a QP from a File

Identifying Convex Quadratic Programming Problems

Conventionally, a quadratic program (QP) is formulated this way:

Minimize $\frac{1}{2}\boldsymbol{x}^T\boldsymbol{Q}x + \boldsymbol{c}^T\boldsymbol{x}$

subject to $\boldsymbol{Ax} \sim b$

with these bounds $l \leq \boldsymbol{x} \leq u$

where the relation ~ may be any combination of equal to, less than or equal to, greater than or equal to, or range constraints. As in other problem formulations, l indicates lower and u upper bounds. $\boldsymbol{Q}$ is a matrix of objective function coefficients. That is, the elements Q_{jj} are the coefficients of the quadratic terms x_j^2, and the elements Q_{ij} and Q_{ji} are summed together to be the coefficient of the term $x_i x_j$.

ILOG CPLEX distinguishes two kinds of $\boldsymbol{Q}$ matrices:

- In a *separable* problem, only the diagonal terms of the matrix are defined.
- In a *nonseparable* problem, at least one off-diagonal term of the matrix is nonzero.

ILOG CPLEX optimizes only *convex* quadratic *minimization* problems or equivalently, only *concave* quadratic *maximization* problems, as illustrated in Figure 7.1. For convex QPs, $\boldsymbol{Q}$ must be *positive semi-definite*; that is, the term $\boldsymbol{x}^T\boldsymbol{Qx} \geq 0$ for all $\boldsymbol{x}$, whether or not $\boldsymbol{x}$ is feasible. For concave maximization problems, the requirement is that $\boldsymbol{Q}$ must be *negative semi-definite*; that is, $\boldsymbol{x}^T\boldsymbol{Qx} \leq 0$ for all $\boldsymbol{x}$. In a minimization problem, if $\boldsymbol{Q}$ is separable and positive semi-definite, then $\boldsymbol{Q} \geq 0$.

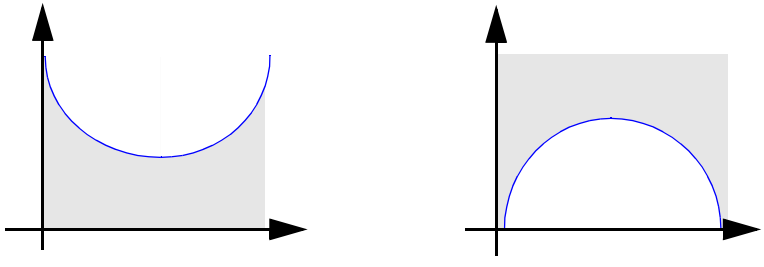

Figure 7.1 *Maximize a Concave Objective Function, Minimize a Convex Objective Function*

In this chapter, we assume you have some familiarity with quadratic programming. For a more complete explanation of quadratic programming generally, we recommend you consult a text such as one of those listed in *Further Reading* on page 30 of the preface of this manual.

Entering QPs

ILOG CPLEX supports two views of quadratic objective functions, a matrix view and an algebraic view.

In the matrix view, commonly found in textbook presentations of QP, the objective function is defined as $^1/_2\, \boldsymbol{x}^T\boldsymbol{Q}x + \boldsymbol{c}^T\boldsymbol{x}$, where $\boldsymbol{Q}$ must be symmetric and positive semi-definite for a minimization problem, or negative semi-definite for a maximization problem. This view is supported by the MPS file format and the Callable Library routines, where the quadratic objective function information is specified by providing matrix $\boldsymbol{Q}$. Thus, by definition, the factor of ½ must be considered when entering a model using the matrix view, as it will be implicitly assumed by the optimization routines. Similarly, symmetry of the $\boldsymbol{Q}$ matrix data is required; the MPS reader will return an error status code if the file contains unequal off-diagonal components, such as a nonzero value for one and zero (or omitted) for the other.

In the algebraic view, a quadratic objective function is specified as an expressions of the form:

```
c1*x1 + ... + cn*xn + q11*x1*x1 + q12*x1*x2 + ... + qnn*xn*xn.
```

This view is supported by the LP format, when entering quadratic objective functions in the Interactive Optimizer, and by Concert Technology. Again, a quadratic objective function must be convex in the case of a minimization problem, or concave in the case of a maximization problem. When entering a quadratic objective with the algebraic view, neither symmetry considerations nor any implicit factors need to be considered, and indeed attempting to specify both of the off-diagonal elements for one of the quadratic terms may result in double the intended value of the coefficient.

Examples:

ILOG CPLEX LP format requires the factor of ½ to be explicitly specified in the file.

```
Minimize
 obj: [ 100 x1 ^2 - 200 x1 * x2 + 100 x2 ^2 ] / 2
```

MPS format for this same objective function would contain the following.

```
QMATRIX
    x1        x1                100
    x1        x2               -100
    x2        x1               -100
    x2        x2                100
```

A C++ Concert program having such an objective function might include the following.

```
model.add(IloMinimize(env, 0.5 * (100*x[0]*x[0] + 100*x[1]*x[1]
                                  - 200*x[0]*x[1] ) ));
```

Or since the algebraic view is supported, the factor of ½ could be simplified as in the following equivalent expression:

```
model.add(IloMinimize(env,   (50*x[0]*x[0] + 50*x[1]*x[1]
                                  - 100*x[0]*x[1] ) ));
```

A similar Java program using Concert might express it this way:

```
IloNumExpr x00 = model.prod(100, model.square(x[0]));
IloNumExpr x11 = model.prod(100, model.square(x[1]));
IloNumExpr x01 = model.prod(-200, model.prod(x[0], x[1]));
IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x01));
model.add(model.minimize(Q));
```

Again, the user could choose to simplify the above expression algebraically if that suits the purposes of the program better.

Finally, a Callable Library program in C might construct the quadratic objective function in a way similar to the following:

```
zqmatind[0] = 0;     zqmatind[2] = 0;
zqmatval[0] = 100.0; zqmatval[2] = -100.0;
zqmatind[1] = 1;     zqmatind[3] = 1;
zqmatval[1] =-100.0; zqmatval[3] = 100.0;
```

To re-emphasize the point about the factor of ½ in any of these methods, if the above objective function is evaluated with a solution of `x1 = 1.000000` and `x2 = 3.000000`, the result to be expected is `200`, not `400`.

Saving QP Problems

After you enter a QP problem, whether interactively or by reading a formatted file, you can then save the problem in a formatted file. The formats available to you are LP, MPS, and SAV. When you save a QP problem in one of these formats, the quadratic information will also be recorded in the formatted file.

In addition, you can save the quadratic part of a problem in a QP file (a formatted file with the extension `.qp`, as described on page 291). To do so:

- In the Interactive Optimizer, use the `write` command
- In the Callable Library, use the routine `CPXqpwrite()`.
- Writing a QP format file is not supported by `IloCplex`.

Changing Problem Type in QPs

Concert Technology (`IloCplex`) programs treat all models as capable of containing quadratic coefficients in the objective function. These coefficients can therefore be added or deleted at will. When extracting a model with a quadratic objective function, `IloCplex` will automatically detect it as a QP and make the required adjustments to data structures.

However, the other ways of using ILOG CPLEX—the Callable Library and the Interactive Optimizer—require an explicit notion of a *Problem Type* to distinguish Linear Programs (LPs) from QPs. This subsection discusses the topic for these users.

When you enter a problem, ILOG CPLEX determines the Problem Type from the available information. When read from a file (LP, MPS, or SAV format, for example), or entered interactively, a continuous optimization problem is treated as being of Type `qp` if quadratic coefficients are present in the objective function. Otherwise, the Problem Type is `lp`. The issue of Problem Types that support integer restrictions in conjunction with quadratic variables is discussed in Chapter 8, *Solving Mixed Integer Programming Problems*.

If you enter a problem that lacks any quadratic coefficients, its Problem Type is initially `lp`. If you then wish to modify the problem to contain quadratic coefficients, you do this by first changing the Problem Type to `qp`. Conversely, if you have entered a QP model and wish to remove all the quadratic coefficients and thus convert the model to an LP, you can change the Problem Type to `lp`. Note that deleting each of the quadratic coefficients individually still leaves the Problem Type as `qp`, although in most instances the distinction between this problem and its `lp` or `qp` counterpart is somewhat arbitrary in terms of the steps to solve it.

When using the Interactive Optimizer, you use the command `change problem` with one of the following options:

- `lp` indicates that you want ILOG CPLEX to treat the problem as an LP. This change in Problem Type removes from your problem all the quadratic information, if there is any present.
- `qp` indicates that you want ILOG CPLEX to treat the problem as a QP. This change in Problem Type creates in your problem an empty quadratic matrix, if there is not one already present, for the objective function, ready for populating via the `change qpterm` command.

From the Callable Library, use the routine `CPXchgprobtype()` to change the Problem Type to either `CPXPROB_LP` or `CPXPROB_QP` for the LP and QP case, respectively, for the same purposes.

Changing Quadratic Terms

ILOG CPLEX distinguishes between a *quadratic algebraic term* and a *quadratic matrix coefficient*. The quadratic algebraic terms are the coefficients that appear in the algebraic expression defined as part of the ILOG CPLEX LP format. The quadratic matrix coefficients appear in $\boldsymbol{Q}$. The quadratic coefficient of an off-diagonal term must be distributed within the $\boldsymbol{Q}$ matrix, and it is always one-half the value of the quadratic algebraic term.

To clarify that terminology, consider this example:

Minimize $a + b + 1/2(a^2 + 4ab + 7b^2)$

subject to $a + b \geq 10$

with these bounds $a \geq 0$ and $b \geq 0$

The off-diagonal quadratic algebraic term in that example is 4, so the quadratic matrix $\boldsymbol{Q}$ is

$$\begin{bmatrix} 1 & 2 \\ 2 & 7 \end{bmatrix}$$

- In a QP, you can change the quadratic matrix coefficients in the Interactive Optimizer by using the command `change qpterm`.
- From the Callable Library, use the routine `CPXchgqpcoef()` to change quadratic matrix coefficients.
- Concert Technology does not support direct editing of expressions other than linear expressions. Consequently, to change a quadratic objective function, you need to create an expression with the modified quadratic objective and use the `setExpr()` method of `IloObjective` to install.

Changing an off-diagonal element changes the corresponding symmetric element as well. In other words, if a call to `CPXchgqpcoef()` changes $\boldsymbol{Q_{ij}}$ to a value, it also changes $\boldsymbol{Q_{ji}}$ to that same value.

To continue our example, if we want to change the off-diagonal quadratic term from 4 to 6, we would use this sequence of commands in the Interactive Optimizer:

```
CPLEX> change qpterm
Change which quadratic term [‘variable’ ‘variable’]: a b
Present quadratic term of variable ‘a’, variable ‘b’ is 4.000000.
Change quadratic term of variable ‘a’, variable ‘b’ to what: 6.0
Quadratic term of variable ‘a’, variable ‘b’ changed to 6.000000.
```

From the Callable Library, the `CPXchgqpcoef()` call to change the off-diagonal term from `4` to `6` would change both of the off-diagonal matrix coefficients from `2` to `3`. Thus, the indices would be `0` and `1`, and the new matrix coefficient value would be 3.

If you have entered a linear problem without any quadratic terms, and you want to create quadratic terms, you must first *change the Problem Type* to QP. To do so, use the command `change problem qp`. This command will create an empty quadratic matrix with $\boldsymbol{Q} = \mathbf{0}$.

When you change quadratic terms, there are still restrictions on the properties of the $\boldsymbol{Q}$ matrix. In a minimization problem, it must be convex, positive semi-definite. In a maximization problem, it must be concave, negative semi-definite. For example, if you change the sense of an objective function in a convex $\boldsymbol{Q}$ matrix from minimization to maximization, you will thus make the problem unsolvable. Likewise, in a convex $\boldsymbol{Q}$ matrix, if you make a diagonal term negative, you will thus make the problem unsolvable.

Optimizing QPs

ILOG CPLEX allows you to solve your QP models through a simple interface, by calling the default optimizer as follows:

- In the Interactive Optimizer, use the command `optimize`.
- From the Callable Library, use the routine `CPXqpopt()`.
- In Concert programs, use the `solve()` method of `IloCplex`.

Under default settings, this will result in the Barrier optimizer being called when solving a continuous QP.

For users who wish to do some performance tuning of their applications, there are two Simplex optimizers that can be tried for solving QPs. They are Dual Simplex and Primal Simplex. You can also use the Network Simplex optimizer; this solves the model as an LP network (temporarily ignoring the quadratic term in the objective function) and takes this solution as a starting point for the Primal Simplex QP optimizer. This choice of QP optimizer is controlled by the `RootAlg` parameter (QPMETHOD in the Interactive Optimizer and in the Callable Library). Permitted values are:

Table 7.1 *RootAlg Parameter Values*

RootAlg Value	Optimizer
0	Automatic (default)
1	Primal Simplex
2	Dual Simplex
3	Network Simplex
4	Barrier

Many of the optimizer tuning decisions for LP apply in the QP case; and parameters that control Barrier and Simplex optimizers in the LP case can be set for the QP case, although in some instances to differing effect. Most models are solved fastest under default parameter settings. See the LP chapter for tuning advice.

Just as for the LP case, each of the available QP optimizers automatically preprocesses your model, conducting presolution problem analysis and reductions appropriate for a QP.

Note that, unlike the LP case, the Barrier optimizer for QP does not provide the capability of a crossover step that provides an optimal *basic* solution. The QP Simplex optimizers, by contrast, do return basic solutions, and these bases can be used for purposes of restarting sequences of optimizations, for instance. As a result, application writers who wish to allow

end users control over the choice of QP optimizer need to be aware of this fundamental difference and to program carefully. For most purposes, the nonbasic Barrier solution is entirely satisfactory, in that all such solutions fully satisfy the standard optimality and feasibility conditions of optimization theory.

Diagnosing QP Infeasibility

Diagnosis of an infeasible QP problem is best done by relaxing it to its associated LP problem, that is, where the ***Q*** matrix has been set to **0**.

To change the Problem Type:

- in the Interactive Optimizer, use the command `change problem lp;`
- from the Callable Library, use the routine `CPXchgprobtype()`.

Since `IloCplex` handles Problem Types transparently, the way to diagnose an infeasible model is slightly different. Since infeasibility does not depend on the objective function, you start by removing the objective extractable from the extracted model. This way, the model seen by the `cplex` object is an LP with a 0 objective and the LP IIS finder can be applied. To get the original model back, simply add the objective back to the model.

You can then solve the relaxed linear version by means of the ILOG CPLEX primal simplex optimizer. Then you can apply the ILOG CPLEX infeasibility finder to that relaxed solution, with its associated, original QP information, to help you diagnose the source of the infeasibility. (*Diagnosing LP Infeasibility* on page 139 explains how to use the ILOG CPLEX infeasibility finder following a simplex optimizer.)

Note that it is possible for the outcome of the above analysis to be a confirmation that your model (viewed as an LP) is feasible after all. This is typically a symptom that your QP model is numerically unstable, or ill-conditioned. Unlike the simplex optimizers for LP, the QP optimizers are primal-dual in nature, and one result of that is the scaling of the objective function interacts directly with the scaling of the constraints.

Just as our recommendation regarding numerical difficulties on LP models (see *Numerical Difficulties* on page 135) is for coefficients in the constraint matrix not to vary by more than about six orders of magnitude, for QP this recommendation expands to include the quadratic elements of the objective function coefficients as well. Fortunately, in most instances it is straightforward to scale your objective function, by multiplying or dividing all the coefficients (linear **and** quadratic) by a constant factor, which changes the unit of measurement for the objective but does not alter the meaning of the variables or the sense of the problem as a whole. If your objective function itself contains a wide variation of coefficient magnitudes, you may also want to consider scaling the individual columns to achieve a closer range.

Example: Creating a QP, Optimizing, Finding a Solution

This example shows you how to build and solve a QP. The problem being created and solved is:

Maximize

$$x_1 + 2x_2 + 3x_3 - 0.5\,(33x_1^2 + 22x_2^2 + 11x_3^2 - 12x_1x_2 - 23x_2x_3)$$

subject to

$$-x_1 + x_2 + x_3 \leq 20$$

$$x_1 - 3x_2 + x_3 \leq 30$$

with these bounds

$$0 \leq x_1 \leq 40$$

$$0 \leq x_2 \leq +\infty$$

$$0 \leq x_3 \leq +\infty$$

Example: iloqpex1.cpp

This example is almost identical to `ilolpex1.cpp` with only function `populatebyrow` to create the model. Also, this function differs only in the creation of the objective from its `ilolpex1.cpp` counterpart. Here the objective function is created and added to the model like this:

```
model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
              - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] + 11*x[2]*x[2]
                           - 12*x[0]*x[1] - 23*x[1]*x[2]) ));
```

In general, any expression built of basic operations `+`, `-`, `*`, `/` constant, and brackets '`()`' that amounts to a quadratic and optional linear term can be used for building QP objective function. Note that, if the expressions of the objective or any constraint of the model contains `IloPiecewiseLinear`, then when a quadratic objective is specified the model becomes an MIQP problem.

Complete Program: iloqpex1.cpp

The complete program, `iloqpex1.cpp`, appears here and online in the standard distribution.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/iloqpex1.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
```

```
//
// iloqpex1.cpp - Entering and optimizing a quadratic problem.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void
   populatebyrow     (IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);
      IloNumVarArray var(env);
      IloRangeArray con(env);

      populatebyrow (model, var, con);

      IloCplex cplex(model);

      // Optimize the problem and obtain solution.
      if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP" << endl;
         throw(-1);
      }

      IloNumArray vals(env);
      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, con);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("qpex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();

   return 0;
}  // END main


// To populate by row, we first create the variables, and then use them to
// create the range constraints and objective.  The model we create is:
//
```

```
//    Maximize
//     obj: x1 + 2 x2 + 3 x3
//            - 0.5 ( 33*x1*x1 + 22*x2*x2 + 11*x3*x3
//                              - 12*x1*x2 - 23*x2*x3 )
//    Subject To
//     c1: - x1 + x2 + x3 <= 20
//     c2: x1 - 3 x2 + x3 <= 30
//    Bounds
//     0 <= x1 <= 40
//    End

static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2]
                            - 0.5 * (33*x[0]*x[0] + 22*x[1]*x[1] +
                                     11*x[2]*x[2] - 12*x[0]*x[1] -
                                     23*x[1]*x[2]                ) ));

   c.add( - x[0] +     x[1] + x[2] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2] <= 30);
   model.add(c);
}  // END populatebyrow
```

Example: QPex1.java

This example is almost identical to `LPex1.java` using only the function `populatebyrow()` to create the model. Also, this function differs only in the creation of the objective from its `LPex1.java` counterpart. Here the objective function is created and added to the model like this:

```
IloNumExpr x00 = model.prod( 33, model.square(x[0]));
IloNumExpr x11 = model.prod( 22, model.square(x[1]));
IloNumExpr x22 = model.prod( 11, model.square(x[2]));
IloNumExpr x01 = model.prod(-12, model.prod(x[0], x[1]));
IloNumExpr x12 = model.prod(-23, model.prod(x[1], x[2]));
IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x22, x01, x12));

double[] objvals = {1.0, 2.0, 3.0};
model.add(model.maximize(model.diff(model.scalProd(x, objvals), Q)));
```

A quadratic objective may be built with `square`, `prod` or `sum` methods. Note that inclusion of `IloPiecewiseLinear` will change the model from a QP to a MIQP.

Complete Program: QPex1.java

```
// --------------------------------------------------------------------------
// File: examples/src/QPex1.java
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 2001-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// QPex1.java - Entering and optimizing a QP problem

import ilog.concert.*;
import ilog.cplex.*;


public class QPex1 {
   public static void main(String[] args) {
      try {
         IloCplex cplex = new IloCplex();
         IloLPMatrix lp = populateByRow(cplex);

         if ( cplex.solve() ) {
            double[] x     = cplex.getValues(lp);
            double[] dj    = cplex.getReducedCosts(lp);
            double[] pi    = cplex.getDuals(lp);
            double[] slack = cplex.getSlacks(lp);

            System.out.println("Solution status = " + cplex.getStatus());
            System.out.println("Solution value  = " + cplex.getObjValue());

            int ncols = lp.getNcols();
            for (int j = 0; j < ncols; ++j) {
               System.out.println("Column: " + j +
                                  " Value = " + x[j] +
                                  " Reduced cost = " + dj[j]);
            }

            int nrows = lp.getNrows();
            for (int i = 0; i < nrows; ++i) {
               System.out.println("Row   : " + i +
                                  " Slack = " + slack[i] +
                                  " Pi = " + pi[i]);
            }

            cplex.exportModel("qpex1.lp");
         }
         cplex.end();
      }
      catch (IloException e) {
         System.err.println("Concert exception '" + e + "' caught");
      }
   }

   static IloLPMatrix populateByRow(IloMPModeler model) throws IloException {
      IloLPMatrix lp = model.addLPMatrix();
```

```
      double[]    lb = {0.0, 0.0, 0.0};
      double[]    ub = {40.0, Double.MAX_VALUE, Double.MAX_VALUE};
      IloNumVar[] x  = model.numVarArray(model.columnArray(lp, 3), lb, ub);

      double[]   lhs = {-Double.MAX_VALUE, -Double.MAX_VALUE};
      double[]   rhs = {20.0, 30.0};
      double[][] val = { {-1.0,  1.0,  1.0},
                         { 1.0, -3.0,  1.0} };
      int[][]    ind = { {0, 1, 2},
                         {0, 1, 2} };
      lp.addRows(lhs, rhs, ind, val);

      // Q = 0.5 ( 33*x0*x0 + 22*x1*x1 + 11*x2*x2 - 12*x0*x1 - 23*x1*x2 )
      IloNumExpr x00 = model.prod( 33, model.square(x[0]));
      IloNumExpr x11 = model.prod( 22, model.square(x[1]));
      IloNumExpr x22 = model.prod( 11, model.square(x[2]));
      IloNumExpr x01 = model.prod(-12, model.prod(x[0], x[1]));
      IloNumExpr x12 = model.prod(-23, model.prod(x[1], x[2]));
      IloNumExpr Q   = model.prod(0.5, model.sum(x00, x11, x22, x01, x12));

      double[] objvals = {1.0, 2.0, 3.0};
      model.add(model.maximize(model.diff(model.scalProd(x, objvals), Q)));

      return (lp);
   }
}
```

Example: qpex1.c

In the routine `setproblemdata()`, there are parameters for `qmatbeg`, `qmatcnt`, `qmatind`, and `qmatval` to fill the quadratic coefficient matrix. The Callable Library routine `CPXcopyquad()` copies this data into the problem object created by the Callable Library routine `CPXcreateprob()`.

In this example, the problem is a *maximization*, so we handle that fact by specifying the objective sense of `CPX_MAX`.

The off-diagonal terms in the matrix $\boldsymbol{Q}$ are one-half the value of the terms x_1x_2, and x_2x_3 as they appear in the algebraic form of the example.

Instead of calling `CPXlpopt()` to find a solution as we do for the *linear* programming problem in `lpex1.c`, this time we call `CPXqpopt()` to optimize this *quadratic* programming problem.

Complete Program: qpex1.c

The complete program, `qpex1.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/qpex1.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/
```

```
/* qpex1.c - Entering and optimizing a quadratic programming problem */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with include of cplex.h. */

#include <ilcplex/cplex.h>
#include <stdlib.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p, double **lb_p,
                   double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                   int **qmatind_p, double **qmatval_p);

static void
   free_and_null (char **ptr);


/* The problem we are optimizing will have 2 rows, 3 columns,
   6 nonzeros, and 7 nonzeros in the quadratic coefficient matrix. */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6
#define NUMQNZ     7


int
main (void)
{
   /* Declare pointers for the variables and arrays that will contain
      the data which define the LP problem.  The setproblemdata() routine
      allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   int      *qmatbeg = NULL;
   int      *qmatcnt = NULL;
   int      *qmatind = NULL;
```

```
double   *qmatval = NULL;

/* Declare and allocate space for the variables and arrays where we
   will store the optimization results including the status, objective
   value, variable values, dual values, row slacks and variable
   reduced costs. */

int      solstat;
double   objval;
double   x[NUMCOLS];
double   pi[NUMROWS];
double   slack[NUMROWS];
double   dj[NUMCOLS];


CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status;
int           i, j;
int           cur_numrows, cur_numcols;

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind,
                         &matval, &lb, &ub, &qmatbeg, &qmatcnt,
                         &qmatind, &qmatval);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
```

```
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create problem.\n");
   goto TERMINATE;
}

/* Now copy the LP part of the problem data into the lp */

status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                    sense, matbeg, matcnt, matind, matval,
                    lb, ub, NULL);

if ( status ) {
   fprintf (stderr, "Failed to copy problem data.\n");
   goto TERMINATE;
}

status = CPXcopyquad (env, lp, qmatbeg, qmatcnt, qmatind, qmatval);
if ( status ) {
   fprintf (stderr, "Failed to copy quadratic matrix.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXqpopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize QP.\n");
   goto TERMINATE;
}

status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution.\n");
   goto TERMINATE;
}


/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);
printf ("Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using what was passed to CPXcopylp.
```

```
      cur_numrows and cur_numcols store the current number of rows and
      columns, respectively.  */

   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "qpex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
```

```
   free_and_null ((char **) &matbeg);
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &qmatbeg);
   free_and_null ((char **) &qmatcnt);
   free_and_null ((char **) &qmatind);
   free_and_null ((char **) &qmatval);

   return (status);

}  /* END main */


/* This function fills in the data structures for the quadratic program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
              - 0.5 ( 33x1*x1 + 22*x2*x2 + 11*x3*x3
                   -  12*x1*x2 - 23*x2*x3 )
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p, double **lb_p,
                double **ub_p, int **qmatbeg_p, int **qmatcnt_p,
                int **qmatind_p, double **qmatval_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   int      *zqmatbeg = NULL;
   int      *zqmatcnt = NULL;
   int      *zqmatind = NULL;
   double   *zqmatval = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
```

```
zsense    = (char *) malloc (NUMROWS * sizeof(char));
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zqmatbeg  = (int *) malloc (NUMCOLS * sizeof(int));
zqmatcnt  = (int *) malloc (NUMCOLS * sizeof(int));
zqmatind  = (int *) malloc (NUMQNZ * sizeof(int));
zqmatval  = (double *) malloc (NUMQNZ * sizeof(double));

if ( zprobname == NULL || zobj     == NULL ||
     zrhs      == NULL || zsense   == NULL ||
     zmatbeg   == NULL || zmatcnt  == NULL ||
     zmatind   == NULL || zmatval  == NULL ||
     zlb       == NULL || zub      == NULL ||
     zqmatbeg  == NULL || zqmatcnt == NULL ||
     zqmatind  == NULL || zqmatval == NULL  )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0]  = 1.0;   zobj[1]   = 2.0;   zobj[2] = 3.0;

zmatbeg[0] = 0;     zmatbeg[1] = 2;    zmatbeg[2] = 4;
zmatcnt[0] = 2;     zmatcnt[1] = 2;    zmatcnt[2] = 2;

zmatind[0] = 0;     zmatind[2] = 0;    zmatind[4] = 0;     zsense[0] = 'L';
zmatval[0] = -1.0;  zmatval[2] = 1.0;  zmatval[4] = 1.0;   zrhs[0]   = 20.0;

zmatind[1] = 1;     zmatind[3] = 1;    zmatind[5] = 1;     zsense[1] = 'L';
zmatval[1] = 1.0;   zmatval[3] = -3.0; zmatval[5] = 1.0;   zrhs[1]   = 30.0;

    zlb[0] = 0.0;       zlb[1] = 0.0;          zlb[2] = 0.0;
    zub[0] = 40.0;      zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND;

/* Now set up the Q matrix.  Note that we set the values knowing that
 * we're doing a maximization problem, so negative values go on
 * the diagonal.  Also, the off diagonal terms are each repeated,
 * by taking the algebraic term and dividing by 2 */

zqmatbeg[0] = 0;     zqmatbeg[1] = 2;     zqmatbeg[2] = 5;
zqmatcnt[0] = 2;     zqmatcnt[1] = 3;     zqmatcnt[2] = 2;

/* Matrix is set up visually.  Note that the x1*x3 term is 0, and is
 * left out of the matrix.  */

zqmatind[0] = 0;     zqmatind[2] = 0;
zqmatval[0] = -33.0; zqmatval[2] = 6.0;

zqmatind[1] = 1;     zqmatind[3] = 1;     zqmatind[5] = 1;
```

```
   zqmatval[1] = 6.0;   zqmatval[3] = -22.0; zqmatval[5] = 11.5;

                        zqmatind[4] = 2;     zqmatind[6] = 2;
                        zqmatval[4] = 11.5;  zqmatval[6] = -11.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zqmatbeg);
      free_and_null ((char **) &zqmatcnt);
      free_and_null ((char **) &zqmatind);
      free_and_null ((char **) &zqmatval);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *qmatbeg_p   = zqmatbeg;
      *qmatcnt_p   = zqmatcnt;
      *qmatind_p   = zqmatind;
      *qmatval_p   = zqmatval;
   }
   return (status);

}  /* END setproblemdata */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
```

```
} /* END free_and_null */
```

Example: Reading a QP from a File

This example shows you how to optimize a QP with routines from the ILOG CPLEX Callable Library when the problem data is stored in a file. The example derives from `lpex2.c`, described in the ILOG CPLEX *Getting Started* manual. The Concert versions of this example, `iloqpex2.cpp` and `QPex2.java`, are included online in the standard distribution.

Instead of calling `CPXlpopt()` to find a solution as we do for the linear programming problem in `lpeq2.c`, this time we call `CPXqpopt()` to optimize this quadratic programming problem.

This example also differs in the available algorithms for solving the problem. QP, barrier with crossover, sifting and concurrent are not available for QP.

Like other applications based on the ILOG CPLEX Callable Library, this one begins with calls to `CPXopenCPLEX()` to initialize the ILOG CPLEX environment and to `CPXcreateprob()` to create the problem object. Before it ends, it frees the problem object with a call to `CPXfreeprob()`, and it frees the environment with a call to `CPXcloseCPLEX()`.

Complete Program: qpex2.c

The complete program, `qpex2.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/qpex2.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* qpex2.c - Reading in and optimizing a QP problem */

/* To run this example, command line arguments are required.
   i.e.,   qpex2   filename   method
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
       method   is the optimization method
                   o          default
                   p          primal simplex
                   d          dual   simplex
                   n          network with dual simplex cleanup
```

```
                    b          barrier without crossover
   Example:
       qpex2  example.mps  o
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

static void
   free_and_null (char **ptr),
   usage         (char *progname);


int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      maximum bound violation, variable values, and basis. */

   int      solnstat, solnmethod, solntype;
   double   objval, maxviol;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           j;
   int           cur_numrows, cur_numcols;
   int           method;

   char          *basismsg;

   /* Check the command line arguments */

   if (( argc != 3 )                          ||
       ( strchr ("podbn", argv[2][0]) == NULL )  ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);
```

```
/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

if ( CPXgetprobtype (env, lp) != CPXPROB_QP ) {
   fprintf (stderr, "Input file is not a QP.  Exiting.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

switch (argv[2][0]) {
   case 'o':
```

```
      method = CPX_ALG_AUTOMATIC;
      break;
   case 'p':
      method = CPX_ALG_PRIMAL;
      break;
   case 'd':
      method = CPX_ALG_DUAL;
      break;
   case 'n':
      method = CPX_ALG_NET;
      break;
   case 'b':
      method = CPX_ALG_BARRIER;
      break;
   default:
      method = CPX_ALG_NONE;
      break;
}

status = CPXsetintparam (env, CPX_PARAM_QPMETHOD, method);
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}


status = CPXqpopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize QP.\n");
   goto TERMINATE;
}

solnstat = CPXgetstat (env, lp);

if      ( solnstat == CPX_STAT_UNBOUNDED ) {
   printf ("Model is unbounded\n");
   goto TERMINATE;
}
else if ( solnstat == CPX_STAT_INFEASIBLE ) {
   printf ("Model is infeasible\n");
   goto TERMINATE;
}
else if ( solnstat == CPX_STAT_INForUNBD ) {
   printf ("Model is infeasible or unbounded\n");
   goto TERMINATE;
}

status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution info.\n");
   goto TERMINATE;
}
printf ("Solution status %d, solution method %d\n", solnstat, solnmethod);

if ( solntype == CPX_NO_SOLN ) {
   fprintf (stderr, "Solution not available.\n");
   goto TERMINATE;
```

```
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr, "Failed to obtain objective value.\n");
   goto TERMINATE;
}
printf ("Objective value %.10g.\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numrows and cur_numcols store the
   current number of rows and columns, respectively.  */

cur_numcols = CPXgetnumcols (env, lp);
cur_numrows = CPXgetnumrows (env, lp);

/* Retrieve basis, if one is available */

if ( solntype == CPX_BASIC_SOLN ) {
   cstat = (int *) malloc (cur_numcols*sizeof(int));
   rstat = (int *) malloc (cur_numrows*sizeof(int));
   if ( cstat == NULL || rstat == NULL ) {
      fprintf (stderr, "No memory for basis statuses.\n");
      goto TERMINATE;
   }

  status = CPXgetbase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, "Failed to get basis; error %d.\n", status);
      goto TERMINATE;
   }
}
else {
   printf ("No basis available\n");
}


/* Retrieve solution vector */

x = (double *) malloc (cur_numcols*sizeof(double));
if ( x == NULL ) {
   fprintf (stderr, "No memory for solution.\n");
   goto TERMINATE;
}

status = CPXgetx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain primal solution.\n");
   goto TERMINATE;
}


/* Write out the solution */

for (j = 0; j < cur_numcols; j++) {
   printf ( "Column %d:  Value = %17.10g", j, x[j]);
   if ( cstat != NULL ) {
      switch (cstat[j]) {
```

```
                case CPX_AT_LOWER:
                   basismsg = "Nonbasic at lower bound";
                   break;
                case CPX_BASIC:
                   basismsg = "Basic";
                   break;
                case CPX_AT_UPPER:
                   basismsg = "Nonbasic at upper bound";
                   break;
                case CPX_FREE_SUPER:
                   basismsg = "Superbasic, or free variable at zero";
                   break;
                default:
                   basismsg = "Bad basis status";
                   break;
             }
             printf ("  %s",basismsg);
          }
          printf ("\n");
       }

       /* Display the maximum bound violation. */

       status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS);
       if ( status ) {
          fprintf (stderr, "Failed to obtain bound violation.\n");
          goto TERMINATE;
       }
       printf ("Maximum bound violation = %17.10g\n", maxviol);


    TERMINATE:

       /* Free up the basis and solution */

       free_and_null ((char **) &cstat);
       free_and_null ((char **) &rstat);
       free_and_null ((char **) &x);

       /* Free up the problem, if necessary */

       if ( lp != NULL ) {
          status = CPXfreeprob (env, &lp);
          if ( status ) {
             fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
          }
       }

       /* Free up the CPLEX environment, if necessary */

       if ( env != NULL ) {
          status = CPXcloseCPLEX (&env);

          /* Note that CPXcloseCPLEX produces no output,
             so the only way to see the cause of the error is to use
             CPXgeterrorstring.  For other CPLEX routines, the errors will
             be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */
```

```
      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

C H A P T E R

8

Solving Mixed Integer Programming Problems

The ILOG CPLEX *Mixed Integer Optimizer* enables you to solve models in which one or more variables must take integer solution values. This chapter tells you more about optimizing mixed integer programming (MIP) problems with ILOG CPLEX. It includes sections on:

- Sample: Stating a MIP Problem
- Considering Preliminary Issues
- Using the Mixed Integer Optimizer
- Using Sensitivity Information in a MIP
- Using Special Ordered Sets
- Using Semi-Continuous Variables
- Progress Reports: Interpreting the Node Log
- Troubleshooting MIP Performance Problems
- Example: Optimizing a Basic MIP Problem
- Example: Reading a MIP Problem from a File
- Example: Using SOS and Priority

Sample: Stating a MIP Problem

A mixed integer programming (MIP) problem may contain both integer and continuous variables. If the problem contains an objective function with no quadratic term, (a *linear objective*), then the problem is termed a *Mixed Integer Linear Program* (MILP). If there is a quadratic term in the objective function, the problem is termed a *Mixed Integer Quadratic Program* (MIQP). In ILOG CPLEX documentation, if the discussion pertains specifically to either the MILP or MIQP case, then that term is used. For the majority of topics that pertain equally to MILP and MIQP, the comprehensive term MIP is used.

The integer variables may be restricted to the values *0* (zero) and *1* (one), in which case they are referred to as *binary* variables. Or they may take on any integer values, in which case they are referred to as *general* integer variables. A variable of any MIP that may take either the value 0 or a value between a lower and an upper bound is referred to as *semi-continuous*. A semi-continuous variable that is restricted to integer values is referred to as *semi-integer.* Special Ordered Sets (SOS) are discussed elsewhere in this chapter. Continuous variables in a MIP problem are those which are *not* restricted in any of these ways, and are thus permitted to take any solution value within their (possibly infinite) lower and upper bounds.

In ILOG CPLEX documentation, the comprehensive term *integer variable* means any of the various types just mentioned except for continuous or SOS. The presence or absence of a quadratic term in the objective function for a given variable has no bearing on its being classified as continuous or integer.

The following illustrates a mixed integer programming problem, which is solved in the example program `ilomipex1.cpp` / `mipex1.c`, discussed later in this chapter:

Maximize		x_1	+	$2x_2$	+	$3x_3$	+	x_4			
subject to	-	x_1	+	x_2	+	x_3	+	$10x_4$	$\leq$	20	
		x_1	-	$3x_2$	+	x_3			$\leq$	30	
				x_2			-	$3.5x_4$	$=$	0	
with these bounds		0	$\leq$	x_1	$\leq$	40					
		0	$\leq$	x_2	$\leq$	$+\infty$					
		0	$\leq$	x_3	$\leq$	$+\infty$					
		2	$\leq$	x_4	$\leq$	3					
				x_4	*integer*						

Considering Preliminary Issues

When you are optimizing a MIP, there are a few preliminary issues that you need to consider to get the most out of ILOG CPLEX. The following sections cover such topics as entering variable type, displaying MIPs in the Interactive Optimizer, determining the problem type, and switching to fixed versions of your problem.

Entering MIP Problems

You enter MIPs into ILOG CPLEX as explained in *Put Data in the Problem Object* on page 86, with this additional consideration: you need to indicate which variables are binary, general integer, semi-continuous, and semi-integer, and which are contained in special ordered sets (SOS).

Concert Technology Library users can specify this information by passing a type value to the appropriate constructor when creating the variable. Use `IloNumVar` objects of type `ILOBOOL` (`IloNumVarType.Bool` in Java) for binary variables and of type `ILOINT` (`IloNumVarType.Int` in Java) for general integer variables. Use `IloSemiContVar` objects of type `ILONUM` (`IloNumVarType.Float` in Java) for semi-continuous variables, and `IloSemiContVar` of type `ILOINT` (`IloNumVarType.Int` in Java) for semi-integer variables.

Callable Library users can specify this information through the `CPXcopyctype()` routine.

In the Interactive Optimizer, to indicate binary integers in the context of the `enter` command, type `binaries` on a separate line, followed by the designated binary variables. To indicate general integers, type `generals` on a separate line, followed by the designated general variables. To indicate semi-continuous variables, type `semi-continuous` on a separate line, followed by the designated variables. Semi-integer variables are indicated by being specified as **both** general integer and semi-continuous. The order of these three sections does not matter. To enter the general integer variable of the *Sample: Stating a MIP Problem* on page 224, you type this:

```
generals
x4
```

You may also read MIP data in from a formatted file, just as you do for linear programming problems. Chapter 9, *More About Using ILOG CPLEX* in this manual describes file formats briefly, and the *ILOG CPLEX Reference Manual* documents file formats, such as MPS, LP, and others.

- To read MIP problem data into the Interactive Optimizer, use the `read` command with an option to indicate the file type.

- To read MIP problem data into your application, use the `importModel()` method in the Concert Technology Library or use `CPXreadcopyprob()` in the Callable Library. from the Callable Library.

Displaying MIP Problems

Table 8.1 summarizes display options in the Interactive Optimizer that are specific to MIP problems.

Table 8.1 *Interactive Optimizer Display Options for MIP Problems*

Interactive command	Purpose
`display problem binaries`	lists variables restricted to binary values
`display problem generals`	lists variables restricted to integer values
`display problem semi-continuous`	lists variables of type semi-continuous and semi-integer
`display problem integers`	lists all of the above
`display problem sos`	lists the names of variables in one or more Special Ordered Sets
`display problem stats`	lists LP statistics plus: • binary variable types, if present; • general variable types, if present; • and number of SOS, if present.

In the Concert Technology Library, use one of the accessor methods supplied with the appropriate object class, such as `IloSOS2::getVariables()`. Refer to the *ILOG Concert Technology Reference Manual* for more information.

From the Callable Library, use the routines `CPXgetctype()` and `CPXgetsos()` to access this information.

Changing Problem Type in MIPs

Concert Technology (`IloCplex`) programs treat all models as capable of containing integer variables, and thus these variable declarations may be added or deleted at will. When extracting a model with integer variables, it will automatically detect it as a MIP and make the required adjustments to data structures.

However, the other ways of using ILOG CPLEX, the Callable Library and the Interactive Optimizer, require an explicit notion of a Problem Type to distinguish continuous LPs and

QPs from MIPs. Techniques to determine the Problem Type with the Callable Library and the Interactive Optimizer are discussed in this topic.

When you enter a problem, ILOG CPLEX determines the Problem Type from the available information. If the problem is read from a file (LP, MPS, or SAV format, for example), or entered interactively, the Problem Type is determined as follows:

Table 8.2 *Problem Type Definitions*

Problem Type	No Integer Variables	Has Integer Variables	No Quadratic Terms in the Objective Function	Has Quadratic Terms in the Objective Function
`lp`	X		X	
`qp`	X			X
`milp`		X	X	
`miqp`		X		X

However, if you enter a problem with no integer variables, so that its Problem Type is initially `lp` or `qp`, and you then wish to modify the problem to contain integer variables, this is accomplished by first changing the Problem Type to `milp` or `miqp`. Conversely, if you have entered an MILP or MIQP model and wish to remove all the integer declarations and thus convert the model to a continuous formulation, you can change the Problem Type to `lp` or `qp`. Note that deleting each of the integer variable declarations individually still leaves the Problem Type as `milp` or `miqp`, although in most instances the distinction between this problem and its `lp` counterpart is somewhat arbitrary in terms of the steps that will be taken to solve it.

Thus, when using the Interactive Optimizer, you use the command `change problem` with one of the following options:

- `milp` or `miqp`

  indicating that you want ILOG CPLEX to treat the problem as an MILP, or MIQP, respectively. This change in Problem Type makes the model ready for declaration of the integer variables via subsequent `change type` commands. If you change the problem to be an MIQP and there are not already quadratic terms in the objective function, an empty quadratic matrix is created, ready for populating via the `change qpterm` command.

- `lp` or `qp`

  indicating that you want all integer declarations removed from the variables in the problem. If you choose the `qp` problem type and there are not already quadratic terms in

the objective function, an empty quadratic matrix is created, ready for populating via the `change qpterm` command.

From the Callable Library, use the routine `CPXchgprobtype()` to change the Problem Type to `CPXPROB_MILP` or `CPXPROB_MIQP` for the MILP and MIQP case respectively, and then assign integer declarations to the variables through the `CPXcopyctype()` function. Conversely, remove all integer declarations from the problem by using `CPXchgprobtype()` with Problem Type `CPXPROB_LP` or `CPXPROB_QP`.

At the end of a MIP optimization, the optimal values for the variables are directly available. However, you may wish to obtain information about the LP or QP associated with this optimal solution, for instance to know the reduced costs for the continuous variables of the problem at this solution. To do this, you will want to change the problem to be of type Fixed, either `fixed_milp` or `fixed_miqp` for the MILP or MIQP case respectively. The fixed MIP is the continuous problem in which the integer variables are fixed at the values they attained in the best integer solution. After changing the problem type, you can then call any of the continuous optimizers to re-optimize, and then display solution information for the continuous version of the problem. If you then wish to change the problem type back to the associated `milp` or `miqp`, you can do so without loss of information in the model.

Changing Variable Type

The command `change type` adds (or removes) the restriction on a variable that it must be an integer. In the Interactive Optimizer, when you enter the command `change type`, the system prompts you to enter the variable that you want to change, and then it prompts you to enter the type (`c` for continuous, `b` for binary, `i` for general integer, `s` for semi-continuous, `n` for semi-integer).

You can change a variable to binary even if its bounds are not *0* (zero) and *1* (one). However, in such a case, the system issues a warning message at optimization, and the optimization may terminate with a bound violation.

Consequently, in the example that we mentioned (see *Sample: Stating a MIP Problem* on page 224), if we want to make x_4 a binary variable, we should first change the *bounds* on x_4 to *0* and *1*; then we can safely change its type to *binary*.

If you change a variable's type to be semi-continuous or semi-integer, be sure to create both a lower bound and an upper bound for it. These variable types specify that at an optimal solution the value for the variable must be either exactly zero or else be between the lower and upper bounds (and further subject to the restriction that the value be an integer, in the case of semi-integer variables).

By the way, a problem may be changed to a mixed integer problem, even if all its variables are continuous.

Using the Mixed Integer Optimizer

The ILOG CPLEX Mixed Integer Optimizer solves MIP models using a very general and robust branch & cut algorithm. While MIP models have the potential to be much more difficult than their continuous LP and QP counterparts, it is also the case that large MIP models are routinely solved in many production applications. A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models. Therefore, it is recommended to try solving your model by first calling the Mixed Integer Optimizer in its most straightforward form.

To invoke the Mixed Integer Optimizer:

- In the Interactive Optimizer, use the `mipopt` command.
- In the Concert Technology Library, with the `IloCplex` method `solve()`.
- In the Callable Library, use the `CPXmipopt()` routine.

Feasibility and Optimality

The following section, *Performance Features of the Mixed Integer Optimizer*, goes into great detail about the algorithmic features, controlled by parameter settings, that are available in ILOG CPLEX to achieve performance tuning on difficult MIP models. However, there is an important parameter, `MIPEmphasis`, that is oriented less toward the user understanding the algorithm being used to solve the model, and more toward the user telling the algorithm something about the underlying aim of the optimization being run. That parameter is discussed here.

Optimizing a MIP model involves:

1. finding a succession of improving integer feasible solutions (solutions satisfying the linear constraints and the integrality conditions); while
2. also working toward a proof that no better feasible solution exists and is undiscovered.

For most models, a balance between these two sometimes-competing aims works well, and this is another way of stating the philosophy behind the default `MIPEmphasis` setting: "balance optimality and integer feasibility".

At this default `MIPEmphasis` setting of `0` (`MIPEmphasisBalanced` or `CPX_MIPEMPHASIS_BALANCED`), ILOG CPLEX uses tactics intended to quickly find a proven optimal solution, for models of a broad range of difficulty. That is, considerable analysis of the model is performed before branching ever begins, in the expectation that the investment will result in a faster total run time, yet not every possible analysis is performed. And then branching is performed in a manner that seeks to find good quality feasible

solutions, without sacrificing too much time that could be spent proving the optimality of any solution that has already been found.

In many situations, the user will desire a greater emphasis on feasibility and less emphasis on analysis and proof of optimality. For instance, a restrictive time limit (set by the user using the `TiLim` parameter) may be in force due to a real-time application deployment, where a model is of sufficient difficulty that a proof of optimality is unlikely, and the user wants to have simply as good a solution as is practicable when the time limit is reached. The `MIPEmphasis` setting of `1` (`MIPEmphasisFeasibility` or `CPX_MIPEMPHASIS_FEASIBILITY`) directs ILOG CPLEX to adopt this emphasis. Less computational effort is applied at the outset toward the analyses that aid in the eventual proof of optimality, and more effort is spent in immediately beginning computations that search for early (and then improved) feasible solutions. It is likely on most models that an eventual proof of optimality would take longer by setting `MIPEmphasis` to `1`, but since the user has given ILOG CPLEX the additional information that this proof is of less importance than usual, the user's needs will actually be met more effectively.

Another choice for `MIPEmphasis` is `2` (`MIPEmphasisOptimality` or `CPX_MIPEMPHASIS_OPTIMALITY`), which results in a greater emphasis on optimality than on feasibility. The search for feasible solutions is not ignored completely, but the balance is shifted toward moving the Best Bound (described in the following paragraph) more rapidly, at the likely expense of feasible solutions being found less rapidly, and improved feasible solutions less frequently, than under the default emphasis. This setting also provides a general (though not precise) compatibility mode to prior versions of ILOG CPLEX.

The fourth choice for `MIPEmphasis`, `3` (`MIPEmphasisBestBound` or `CPX_MIPEMPHASIS_BESTBOUND`), works exclusively at moving the *Best Bound*. The Best Bound represents the objective function value at which an integer feasible solution could still potentially exist. As possibilities are eliminated, this Best Bound value will move in the opposite direction to that of any improving series of integer feasible solutions. The process of moving the Best Bound will eventually result in the optimal feasible solution being discovered, at which point the optimization is complete, and feasible solutions may be discovered along the way anyway, due to branches that happen to locate feasible solutions that do not match the Best Bound. A great deal of analysis may be performed on the model, beyond what is done under the default emphasis. Therefore it is recommended to use this setting only on models that are difficult for the default emphasis, and for which you do not care about interim feasible solutions that may or may not be optimal.

To make clear a point that has been alluded to in the above: every choice of `MIPEmphasis` results in the branch & cut algorithm proceeding in a manner that eventually will find and prove an optimal solution, or will prove that no integer feasible solution exists. The choice of emphasis only guides ILOG CPLEX to produce feasible solutions in a way that is in keeping with the user's particular purposes, but the accuracy and completeness of the algorithm is not sacrificed in the process.

The MIPEmphasis parameter may be set in conjunction with any other ILOG CPLEX parameters (discussed at length in the next section). For example if you wish to set an upward branching strategy via the BrDir parameter, this will be honored under any setting of MIPEmphasis. Of course, certain combinations of MIPEmphasis with other parameters may be counter-productive, such as turning off all Cuts under emphasis 3, but the user has the option if that is what is desired.

Performance Features of the Mixed Integer Optimizer

The ILOG CPLEX Mixed Integer Optimizer contains a wealth of features intended to aid in the solution of challenging MIP models. While default strategies are provided that solve the majority of models without user involvement, there exist difficult models that benefit from attention to performance tuning. This section discusses the ILOG CPLEX parameters that are the most likely to offer help on such models.

Because many of these parameter settings directly affect the branch & cut algorithm, we first give a description of how that algorithm is implemented within ILOG CPLEX.

Branch & Cut

In the branch & cut algorithm, ILOG CPLEX solves a series of continuous subproblems. To manage those subproblems efficiently, ILOG CPLEX builds a tree in which each subproblem is a node. The root of the tree is the *continuous relaxation* of the original MIP problem.

If the solution to the relaxation has one or more fractional variables, ILOG CPLEX will try to find cuts. Cuts are constraints that cut away areas of the feasible region of the relaxation that contain fractional solutions. ILOG CPLEX can generate several types of cuts. (*Cuts* on page 235 tells you more about that topic.)

If the solution to the relaxation still has one or more fractional-valued integer variables after ILOG CPLEX tries to add cuts, then ILOG CPLEX branches on a fractional variable to generate two new subproblems, each with more restrictive bounds on the branching variable. For example, with binary variables, one node will fix the variable at *0* (zero), the other, at *1* (one).

The subproblems may result in an all-integer solution, in an infeasible solution, or another fractional solution. If the solution is fractional, ILOG CPLEX repeats the process.

ILOG CPLEX cuts off nodes when the value of the objective function associated with the subproblem at that node is worse than the cutoff value.

You set the cutoff value by means of the CutUp parameter (for a minimization problem) or the CutLo parameter (for a maximization problem), to indicate to ILOG CPLEX that integer feasible solutions worse than this cutoff value should be discarded. The default value of the lower cutoff is $-1e^{+75}$; the default value of the upper cutoff is $1e^{+75}$. The defaults, in effect, mean that no cutoff is to be supplied. You can supply any number that you find appropriate

for your problem. It is never required that you supply a cutoff, and in fact for most applications is it not done.

ILOG CPLEX will use the value of the best integer solution found so far, as modified by the tolerance parameters `ObjDif` or `RelObjDif` (absolute and relative objective function difference, respectively), as the cutoff. Again, it is not typical that users set these parameters, but they are available if you find them useful. Use care in changing these tolerances: if either of them is nonzero, you may miss the optimal solution by as much as that amount. For example, in a model where the true minimum is `100` and the absolute cutoff is set to `5`, if a feasible solution of say, `103` is found at some point, the cutoff will discard all nodes with a solution worse than `98`, and thus the solution of `100` would be overlooked.

Periodically during the branch & cut algorithm, ILOG CPLEX may apply a heuristic process that attempts to compute an integer solution from available information, such as the solution to the relaxation at the current node. This activity does not replace the branching steps, but sometimes is able to inexpensively locate a new feasible solution sooner than by branching, and a solution found in this way is treated in the same way as any other feasible solution. At intervals in the tree, new cuts beyond those computed at the root node may also be added to the problem.

Once ILOG CPLEX finds an integer solution, it does the following:

- it makes that integer solution the *incumbent solution* and that node the *incumbent node*;
- it makes the value of the objective function at that node (modified by the objective difference parameter) the new cutoff value;
- it prunes from the tree all subproblems for which the value of the objective function is no better than the incumbent.

You control the path that CPLEX traverses in the tree through several parameters, as summarized in Table 8.3.

Table 8.3 *Parameters for Controlling Branch & Cut Strategy*

Interactive Optimizer Command	Concert Technology IloCplex Method	Callable Library Routine
`set mip strategy backtrack`	`setParam(BtTol, n)`	`CPXsetdblparam(env, CPX_PARAM_BTTOL, n)`
`set mip strategy nodeselect`	`setParam(NodeSel, i)`	`CPXsetintparam(env, CPX_PARAM_NODESEL, i)`
`set mip strategy variableselect`	`setParam(VarSel, i)`	`CPXsetintparam(env, CPX_PARAM_VARSEL, i)`
`set mip strategy bbinterval`	`setParam(BBInterval, i)`	`CPXsetintparam(env, CPX_PARAM_BBINTERVAL, i)`
`set mip strategy branch`	`setParam(BrDir, i)`	`CPXsetintparam(env, CPX_PARAM_BRDIR, i)`

During the branch & cut algorithm, ILOG CPLEX may choose to continue from the present node and dive deeper into the tree, or it may backtrack (that is, begin a new dive from elsewhere in the tree). The value of the backtrack parameter, `BtTol`, influences this

decision, in terms of the relative degradation of the objective function caused by the branches taken so far in this dive. Setting `BtTol` to a value near `0.0` increases the likelihood that a backtrack will occur, while the default value near `1.0` makes it more likely that the present dive will continue to a resolution (fathoming either via a cutoff or an infeasible combination of branches or the discovery of a new incumbent integer feasible solution). See the *ILOG CPLEX Reference Manual*, *Appendix A*, for more details on how this parameter affects the computation that determines the decision to backtrack.

When ILOG CPLEX backtracks, there usually remain large numbers of unexplored nodes from which to begin a new dive. The node selection parameter, `NodeSel`, determines this choice.

Table 8.4 *NodeSel Parameter Values for Node Search Type*

NodeSel Value	Symbolic Value	Node Search Type
1 (Default)	CPX_NODESEL_BESTBOUND	***Best Bound*** search, which means that the node with the best objective function will be selected, generally near the top of the tree.
2	CPX_NODESEL_BESTEST	***Best Estimate*** search, whereby ILOG CPLEX will estimate a given node's progress toward integer feasibility relative to its degradation of the objective function. This setting can be useful in cases where there is difficulty in finding feasible solutions or in cases where a proof of optimality is not crucial.
2	CPX_NODESEL_BESTEST_ALT	A variation on the Best Estimate search.
0	CPX_NODESEL_DFS	***Depth First*** search will be conducted. In many cases this amounts to a brute force strategy for solving the combinatorial problem, gaining a small amount of tactical efficiency due to a variety of reasons, and it is rare that it offers any advantage over other settings.

In instances where Best Estimate node selection (`NodeSel` = 2 or 3) is in effect, the `BBInterval` parameter determines the frequency at which backtracking is done by Best Bound anyway. The default value of `7` works well, but you can set it to `0` to assure that Best Estimate is used every time backtracking occurs.

Once a node has been selected, the variable selection parameter, `VarSel`, influences which variable is chosen for branching at that node.

Table 8.5 *VarSel Parameter Values for Branching Variable Choice*

VarSel Value	Symbolic Value	Branching Variable Choice
-1	CPX_VARSEL_MININFEAS	Branch strictly at the nearest integer value which is closest to the fractional variable.
1	CPX_VARSEL_MAXINFEAS	Branch strictly at the nearest integer value which is furthest from the fractional variable.
0 (Default)	CPX_VARSEL_DEFAULT	ILOG CPLEX makes the determination of each branch direction.
2	CPX_VARSEL_PSEUDO	Use pseudo costs, which derives an estimate about the effect of each proposed branch from duality information.
3	CPX_VARSEL_STRONG	Use Strong Branching, which invests considerable effort in analyzing potential branches in the hope of drastically reducing the number of nodes that will be explored.
4	CPX_VARSEL_PSEUDOREDUCED	Use pseudo reduced costs, which is a computationally less-intensive version of pseudo costs.

Once a variable has been selected for branching, the `BrDir` parameter influences the direction, up or down, of the branch on that variable to be explored first.

Table 8.6 *BrDir Parameter Values for Branching Direction Choice*

BrDir Value	Symbolic Value	Branching Direction Choice
0 (Default)	CPX_BRANCH_GLOBAL	ILOG CPLEX makes the determination of each branch direction.
-1	CPX_BRANCH_DOWN	Branch downward.
1	CPX_BRANCH_UP	Branch upward.

Note that *Priority Orders* on page 240, complement the behavior of these parameters, as a mechanism by which you supply problem-specific directives about the order in which to branch on variables. In a Priority Order you can also provide preferred branching directions for specific variables.

Cuts

Cuts are constraints added to a model to restrict (cut away) noninteger solutions that would otherwise be solutions of the continuous relaxation. The addition of cuts usually reduces the number of branches needed to solve a MIP.

In the following descriptions of cuts, the term *subproblem* includes the root node (that is, the root relaxation). Cuts are most frequently seen at the root node, but they may be added by ILOG CPLEX at other nodes as conditions warrant.

ILOG CPLEX generates its cuts in such a way that they are valid for all subproblems, even when they are discovered during analysis of a particular subproblem. If the solution to a subproblem violates one of the subsequent cuts, ILOG CPLEX may add a constraint to reflect this condition.

Clique Cuts

A *clique* is a relationship among a group of binary variables such that at most one variable in the group can be positive in any integer feasible solution. Before optimization starts, ILOG CPLEX constructs a graph representing these relationships and finds maximal cliques in the graph.

Cover Cuts

If a constraint takes the form of a *knapsack constraint* (that is, a sum of binary variables with nonnegative coefficients less than or equal to a nonnegative right-hand side), then there is a minimal cover associated with the constraint. A *minimal cover* is a subset of the variables of the inequality such that if all the subset variables were set to one, the knapsack constraint would be violated, but if any one subset variable were excluded, the constraint would be satisfied. ILOG CPLEX can generate a constraint corresponding to this condition, and this cut is called a *cover cut*.

Disjunctive Cuts

A MIP problem can be divided into two subproblems with disjunctive feasible regions of their LP relaxations by branching on an integer variable. *Disjunctive cuts* are inequalities valid for the feasible regions of LP relaxations of the subproblems, but not valid for the feasible region of LP relaxation of the MIP problem.

Flow Cover Cuts

Flow covers are generated from constraints that contain continuous variables, where the continuous variables have variable upper bounds that are zero or positive depending on the setting of associated binary variables. The idea of a flow cover comes from considering the constraint containing the continuous variables as describing a single node in a network where the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables for the variable upper bounds. The flows and the demand at the single node imply a knapsack constraint. That knapsack

constraint is then used to generate a cover cut on the flows (that is, on the continuous variables and their variable upper bounds).

Flow Path Cuts

Flow path cuts are generated by considering a set of constraints containing the continuous variables that describe a path structure in a network, where the constraints are nodes and the continuous variables are in-flows and out-flows. The flows will be on or off depending on the settings of the associated binary variables.

Gomory Fractional Cuts

Gomory fractional cuts are generated by applying integer rounding on a pivot row in the optimal LP tableau for a (basic) integer variable with a fractional solution value.

Generalized Upper Bound (GUB) Cover Cuts

A *GUB constraint* for a set of binary variables is a sum of variables less than or equal to one. If the variables in a GUB constraint are also members of a knapsack constraint, then the minimal cover can be selected with the additional consideration that at most one of the members of the GUB constraint can be one in a solution. This additional restriction makes the *GUB cover cuts* stronger (that is, more restrictive) than ordinary cover cuts.

Implied Bound Cuts

In some models, binary variables imply bounds on continuous variables. ILOG CPLEX generates potential cuts to reflect these relationships.

Mixed Integer Rounding (MIR) Cuts

MIR cuts are generated by applying integer rounding on the coefficients of integer variables and the right-hand side of a constraint.

Adding Cuts and Re-Optimizing

Each time ILOG CPLEX adds a cut, the subproblem is re-optimized. ILOG CPLEX repeats the process of adding cuts at a node until it finds no further effective cuts. It then selects the branching variable for the subproblem.

Parameters Affecting Cuts

Parameters control the way each class of cuts is used. Those parameters are listed in Table 8.7.

Table 8.7 *Parameters for Controlling Cuts*

Cut Type	Interactive Command	Concert Technology Library Parameter	Callable Library Parameter
Clique	`set mip cuts cliques`	`Cliques`	`CPX_PARAM_CLIQUES`
Cover	`set mip cuts covers`	`Covers`	`CPX_PARAM_COVERS`
Disjunctive	`set mip cuts disjunctive`	`DisjCuts`	`CPX_PARAM_DISJCUTS`
Flow Cover	`set mip cuts flowcuts`	`FlowCovers`	`CPX_PARAM_FLOWCOVERS`
Flow Path	`set mip cuts pathcut`	`FlowPaths`	`CPX_PARAM_FLOWPATHS`
Gomory	`set mip cuts gomory`	`FracCuts`	`CPX_PARAM_FRACCUTS`
GUB Cover	`set mip cuts gubcovers`	`GUBCovers`	`CPX_PARAM_GUBCOVERS`
Implied Bound	`set mip cuts implied`	`ImplBd`	`CPX_PARAM_IMPLBD`
Mixed Integer Rounding (MIR)	`set mip cuts mircut`	`MIRCuts`	`CPX_PARAM_MIRCUTS`

The default value of each of those parameters is `0` (zero). By default, ILOG CPLEX automatically determines how often (if at all) it should try to generate that class of cut. A setting of `-1` indicates that no cuts of the class should be generated; a setting of `1` indicates that cuts of the class should be generated moderately; and a setting of `2` indicates that cuts of the class should be generated aggressively. For disjunctive cuts, a setting of 3 is permitted, which indicates that disjunctive cuts should be generated very aggressively.

In the Interactive Optimizer, the command `set mip cuts all` *`i`* applies the value *`i`* to all classes of cut parameters. That is, you can set them all at once.

The `CutsFactor` parameter controls the number of cuts ILOG CPLEX adds to the model. The problem can grow to `CutsFactor` times the original number of rows in the model (or in the presolved model, if the presolver is active). Thus, a `CutsFactor` of 1.0 would mean that no cuts will be generated, which may be a more convenient way of turning off all cuts than setting them individually. The default `CutsFactor` value of 4.0 works well in most cases, as it allows a generous number of cuts while in rare instances it also serves to limit unchecked growth in the problem size.

The AggCutLim parameter controls the number of constraints allowed to be aggregated for generating MIR and flow cover cuts.

The FracPass parameter controls the number of passes for generating Gomory fractional cuts. This parameter will not have any effect if the parameter for set mip cuts gomory has a non-default value.

The FracCand parameter controls the number of variable candidates to be considered for generating Gomory fractional cuts.

Heuristics

ILOG CPLEX provides a heuristic to find integer solutions at nodes during the branch & cut procedure.

This feature is controlled by the parameter HeurFreq. At its default value of 0, ILOG CPLEX dynamically determines the frequency with which the heuristic is invoked. The value of -1 turns the feature off. A positive value specifies the frequency (in node count) with which the heuristic will be called. For example, if the HeurFreq parameter is set to 20, then the node heuristic will be applied at node 0, node 20, node 40, and so on.

Probing

While most of the suggestions in this section are oriented toward overcoming specific obstacles, the *probing* feature can help in many different ways on difficult models. Probing is a technique that looks at the logical implications of fixing each binary variable to 0 or 1. It is performed after preprocessing and before the solution of the root relaxation. Probing can be expensive, so this parameter should be used selectively. On models that are in some sense easy, the extra time spent probing may not reduce the overall time enough to be worthwhile. On difficult models, probing may incur very large runtime costs at the beginning and yet pay off with shorter overall runtime. When you are tuning performance, it is usually because the model is difficult, and then probing is worth trying.

At the Probe parameter's default setting of 0, ILOG CPLEX will automatically determine the appropriate level of probing. Setting the Probe parameter to 1, 2, or 3, results in increasing levels of probing to be performed beyond the default level of probing. A setting of -1 results in no probing being performed.

To activate probing::

- In the Interactive Optimizer, use the command set mip strategy probe i.
- In the Concert Technology Library, set the integer parameter Probe.
- In the Callable Library, set the integer parameter CPX_PARAM_PROBE.

Preprocessing: Presolver and Aggregator

When you invoke the MIP optimizer, whether through the Interactive Optimizer command `mipopt`, through a call to the Concert Technology `IloCplex` method `solve()`, or through the Callable Library routine `CPXmipopt()`—ILOG CPLEX by default automatically preprocesses your problem. Table 8.8 summarizes the preprocessing parameters. In preprocessing, ILOG CPLEX applies its presolver and aggregator one or more times to reduce the size of the integer program in order to strengthen the initial linear relaxation and to decrease the overall size of the mixed integer program.

Table 8.8 *Parameters for Controlling MIP Preprocessing*

Interactive Command	Concert Technology Library Parameter	Callable Library Parameter	Comment
`set preprocessing aggregator`	`AggInd`	`CPX_PARAM_AGGIND`	on by default
`set preprocessing presolve`	`PreInd`	`CPX_PARAM_PREIND`	on by default
`set preprocessing boundstrength`	`BndStrenInd`	`CPX_PARAM_BNDSTRENIND`	presolve must be on
`set preprocessing coeffreduce`	`CoeRedInd`	`CPX_PARAM_COEREDIND`	presolve must be on
`set preprocessing relax`	`RelaxPreInd`	`CPX_PARAM_RELAXPREIND`	applies to relaxation
`set preprocessing reduce`	`Reduce`	`CPX_PARAM_REDUCE`	all on by default
`set preprocessing numpass`	`PrePass`	`CPX_PARAM_PREPASS`	automatic by default

The parameters `Reduce` and `Numpass` have the same meanings for LP, QP and MIP. *Preprocessing* on page 127 explains the meanings and adjustments of all these parameters.

While preprocessing, ILOG CPLEX also attempts to strengthen bounds on variables. This bound strengthening may take a long time. In such cases, you may want to turn off bound strengthening.

ILOG CPLEX also attempts to reduce coefficients during preprocessing. Coefficient reduction usually strengthens the continuous relaxation and reduces the number of nodes in the branch & cut tree, but not always. Sometimes, it increases the amount of time needed to solve the linear relaxations at each node—enough time to offset the benefit of fewer nodes. Two levels of coefficient reduction are available, so it is worthwhile to experiment with these preprocessing options to see whether they are beneficial to your problem.

In addition, you may also set the relaxation parameter to tell ILOG CPLEX to apply preprocessing to the initial relaxation of the problem. Sometimes this preprocessing will result in additional, beneficial presolve transformations in the LP or QP relaxation, transformations that are not possible in the original MIP model. In general, MIP preprocessing reductions and continuous LP or QP reductions are not the same, as some

MIP reductions are not applicable to the continuous case, and some continuous reductions do not apply in the MIP case. Thus, the `RelaxPreInd` parameter allows you to get the benefit of both styles of preprocessing, when the model is large enough that the root solution is itself a time consuming step.

ILOG CPLEX preprocesses a MIP by default. However, if you use a basis to start LP optimization of the root relaxation, ILOG CPLEX will proceed with that starting basis without preprocessing it. Frequently the strategic benefits of MIP presolve outweigh the tactical advantage of using a starting basis for the root node, so use caution when considering the advantages of a starting basis.

Starting from a Solution

You can provide a known solution (for example, from a MIP problem previously solved or from your knowledge of the problem) to serve as the first integer solution. In such a start, you must specify values for all integer variables, for all semi-continuous variables, and for all members of special ordered sets. Optionally, you may also specify values for continuous variables. ILOG CPLEX evaluates that start solution for integrality and feasibility. If it is integer-feasible, it will become an integer solution of the current problem.

Once a MIP Start has been established for your model, its use is controlled by the `MIPStart` parameter. At its default setting of `0`, the MIP Start values are not used, but you can direct ILOG CPLEX to evaluate the MIP Start values by setting the `MIPStart` parameter to `1`.

You can establish MIP starting values by using the method `setVectors()` in a Concert program, or by using `CPXcopymipstart()` in a Callable Library program.

For Interactive Optimizer usage, or as an alternative approach in a Callable Library program, you can establish MIP starting values from a file. `MST` format (described briefly in *Understanding File Formats* on page 290 or documented in the *ILOG CPLEX Reference Manual*) is used for this purpose. Use `CPXreadcopymipstart()` in the Callable Library, or the "`read`" command in the Interactive optimizer, for this purpose.

At the end of a MIP optimization call, when a feasible (not necessarily optimal) solution is still in memory, you can create an `MST` file from the Callable Library using `CPXmstwrite()`, or from the Interactive Optimizer using the `write` command, for later use as starting values to another MIP problem. Care should be taken to assure that the naming convention for the variables is consistent between models, when this approach is used.

Priority Orders

In branch & cut, ILOG CPLEX makes decisions about which variable to branch on at a node. You can control the order in which ILOG CPLEX branches on variables by issuing a *priority order*. A priority order assigns a branching priority to some or all of the integer variables in a model. ILOG CPLEX performs branches on variables with a higher assigned priority number before variables with a lower priority; variables not assigned an explicit

priority value by the user are treated as having a priority value of 0. Note that ILOG CPLEX will branch only on variables that take a fractional solution value at a given node. Thus a variable with a high priority number might still not be branched upon until late in the tree, if at all, and indeed if the presolve or the aggregator feature of the ILOG CPLEX Preprocessor removes a given variable then branching on that variable would never occur regardless of a high priority order assigned to it by the user.

You can specify priority for any variable, though the priority is used only if the variable is a general integer variable, a binary integer variable, or a member of a special ordered set.

ILOG CPLEX provides the ability to generate a priority order automatically, based on problem-data characteristics. This facility can be activated by setting the MIPOrdType parameter to one of the following:

Table 8.9 *Parameters for Branching Priority Order*

Parameter	Branching Priority Order
0	no automatic priority order will be generated (default)
1	decreasing cost coefficients among the variables
2	increasing bound range among the variables
3	increasing cost per matrix coefficient count among the variables

If you explicitly read a file of priority orders, its settings will override any generic priority order you may have set by interactive commands.

The parameter MIPOrdInd, when set to 0, allows you to direct ILOG CPLEX to ignore a priority order that was previously read from a file. The default setting for this parameter means that a priority order will be used, if one has been read in.

Problems that use integer variables to represent different types of decisions should assign higher priority to those that must be decided first. For example, if some variables in a model activate processes, and others use those activated processes, then the first group of variables should be assigned higher priority than the second group. In that way, you can use priority to achieve better solutions.

Setting priority based on the magnitude of objective coefficients is also sometimes helpful.

Termination

ILOG CPLEX terminates MIP optimization under a variety of circumstances. First, ILOG CPLEX declares integer optimality and terminates when it finds an integer solution and all nodes have been processed. Optimality in this case is relative to whatever tolerances and optimality criteria you have set. For example, ILOG CPLEX considers the cutoff value and the objective difference parameter in this context.

In addition, ILOG CPLEX terminates optimization when it reaches a limit that you have set. You can set limits on time, number of nodes, size of tree memory, and number of integer solutions. Table 8.10 summarizes those parameters and their purpose.

Table 8.10 *Parameters to limit MIP optimization*

To set a limit on	Use this parameter		
	Concert Technology	Callable Library	Interactive Optimizer
elapsed time	`TiLim`	`CPX_PARAM_TILIM`	`timelimit`
number of nodes	`NodeLim`	`CPX_PARAM_NODELIM`	`mip limits nodes`
size of tree	`TreLim`	`CPX_PARAM_TRELIM`	`mip limits treememory`
size of node file	`WorkMem`	`CPX_PARAM_WORKMEM`	`workmem`
number of integer solutions	`IntSolLim`	`CPX_PARAM_INTSOLLIM`	`mip limits solutions`

ILOG CPLEX also terminates when an error occurs, such as when ILOG CPLEX runs out of memory or when a subproblem cannot be solved. If an error is due to failure to solve a subproblem, an additional line appears in the node log file to indicate the reason for that failure.

Writing a Tree File

When ILOG CPLEX terminates a MIP optimization before it achieves optimality (for example, because it has reached a limit you set), it still has significant information about the current branch & cut tree. You can save this information by writing it to a file of type TRE (a binary, proprietary ILOG CPLEX format). Later, you can then read the saved TRE file and restart the optimization from where ILOG CPLEX left off.

To save a MIP in a TRE file:

- In the Interactive Optimizer, use the command `write filename.tre` or `write filename tre`.
- From the Callable Library, use the routine `CPXtreewrite()`.

Note that resuming a branch & cut optimization by reading a tree file back in may not result in precisely the same solution path being taken as though the run had not been interrupted, due simply to the dynamic nature of the branch & cut algorithm.

A TRE file may be quite large (corresponding to the current size of an active tree) so it may consume considerable disk space.

If you modify the model of a MIP *after* you create its TRE file, then the TRE file will be of no use to you. ILOG CPLEX will accept the old TRE file if the basic dimensions of the problem have not changed, but the results it produces from it will likely be invalid for the modified model.

Post-Solution Information in a MIP

Interpreting Solution Statistics on page 141 explains how to use the `display` command in the Interactive Optimizer to see post-solution information from the *linear* optimizers. However, because of the way integer solutions are generated, the `display` command shows you only limited information from the MIP optimizer. In fact, ILOG CPLEX generates integer solutions by solving subproblems that have different bounds from the original problem, so computing solution values with the original bounds will not usually give the same solution. Nevertheless, the following solution statistics are available from the MIP optimizer:

- objective function value for the best integer solution, if one exists;
- best bound, that is, best objective function value among remaining subproblems;
- solution quality;
- primal values for the best integer solution, if one has been found;
- slack values for best integer solution, if one has been found.

If you request other solution statistics, ILOG CPLEX will issue the error message, “Not available for mixed integer problems—use CHANGE PROBLEM to change the problem type.”

Using Sensitivity Information in a MIP

Other post-solution information does not have the same meaning in a mixed integer program as in a linear program because of the special nature of the integer variables in the MIP. The reduced costs, dual values, and sensitivity ranges give you information about the effect of making small changes in problem data so long as *feasibility is maintained*. Integer variables, however, *lose feasibility* if a small change is made in their value, so this post-solution information cannot be used to evaluate changes in problem data in the usual way of LPs.

Integer variables typically represent major structural decisions in a model, and often many continuous variables of the model are related to these major decisions. With that observation in mind, if you take the integer variable values as given, then you can use post-solution information *applying only to the continuous variables* in the usual way.

To access this limited sensitivity information in a MIP:

- In the Interactive Optimizer, use the command `change problem fixed` to fix the values of the integer variables.
- In the Callable Library, use the routine `CPXchgprobtype()`.

ILOG CPLEX then sets the variable bounds so that upper and lower bounds are those in the current integer solution. You can then optimize the resulting continuous program and display its post-solution statistics.

Using Special Ordered Sets

A special ordered set (SOS) is an additional way to specify integrality conditions in a model. There are various types of SOS:

- SOS Type 1 is a set of variables (whether all integer, all continuous, or mixed integer and continuous) where at most one variable may be nonzero.
- SOS Type 2 is a set or integer or continuous variables where at most two variables may be nonzero. If two variables are nonzero, they must be adjacent in the set.

ILOG CPLEX uses special branching strategies to take advantage of SOSs. The special branching strategies depend upon the order among the variables in the set. The order is specified by assigning weights to each variable. The order of the variables in the model (such as in the MPS or LP format data file, or the column index in a Callable Library application) is ***not*** used in SOS branching. If there is no order relationship among the variables (such that weights cannot be specified or would not be meaningful), SOS branching should not be used. For many classes of problems, these branching strategies can significantly improve performance.

Example: SOS Type 1 for Sizing a Warehouse

To give you a feel for how SOSs can be useful, here's an example of an SOS Type 1 used to choose the size of a warehouse. Let's assume for this example that we can build a warehouse of *10000*, *20000*, *40000*, or *50000* square feet. We define binary variables for the four sizes, say, x_1, x_2, x_4, and x_5. We connect these variables by a constraint defining another variable to denote available square feet, like this: $z - 10000x_1 - 20000x_2 - 40000x_4 - 50000x_5 = 0$.

Those four variables are members of a special ordered set. Only one size can be chosen for the warehouse; that is, at most one of the *x* variables can be nonzero in the solution. And, there is an order relationship among the *x* variables (namely, the sizes) that can be used as weights. We say that the *weights* of the set members are *10000*, *20000*, *40000*, and *50000*.

Let's say furthermore that we have a fractional (that is, noninteger) solution of $x_1 = 0.1$, $x_5 = 0.9$. These values indicate that other parts of the model have imposed the requirement of *46000* square feet since $0.1*10000 + 0.9*50000 = 46000$. In SOS parlance, we say that the *weighted average* of the set is $(0.1*10000 + 0.9*50000)/(0.1 + 0.9) = 46000$.

We *split* the set before the variable with weight exceeding the weighted average. In this case, we split the set like this: x_1, x_2, and x_4 will be in one subset; x_5 in the other.

Now we branch. One branch restricts x_1, x_2, x_4 to *0* (zero). This branch results in x_5 being set to *1* (one).

The other branch, where x_5 is set to *0* (zero), results in an infeasible solution, so we remove it from further consideration.

If a warehouse must be built, then we need the additional constraint that $x_1 + x_2 + x_4 + x_5 = 1$. The implicit constraint for an SOS Type 1 is less than or equal to one. The continuous relaxation may more closely resemble the MIP if we add that constraint.

Declaring SOS Members

ILOG CPLEX offers you several ways to declare an SOS in a problem:

- Use an SOS file, that is one in SOS format, with the file extension `.sos`. SOS files offer you the most powerful and flexible alternative because the SOS file structure allows you to do several tasks at once:
  - provide branching priorities for sets,
  - assign weights to individual set members,
  - define overlapping sets.
- Use SOS declarations within an MPS or LP file (that is, one in MPS format with the file extension `.mps` or in LP format with the file extension `.lp`. If you already have MPS files with SOS information, you may prefer this option. Conventions for declaring SOS information in MPS files are documented in the *ILOG CPLEX Reference Manual*.

Setting Branching Priority for an SOS

An entire SOS can be given a branching priority. There are two alternative ways to give an SOS branching priority, both documented in the *ILOG CPLEX Reference Manual*:

- Use an SOS file to set priorities.
- Use an ORD file to set priorities.

ILOG CPLEX derives the branching priority of a *set* from the branching priorities of its *members*: the entire set is assigned the highest priority among its members.

To specify SOS priorities:

- In the Concert Technology Library, use the functions `IloCplex::setPriority()` and `IloCplex::setPriorities()`.
- In the Callable Library, use the routines `CPXcopysos()` or `CPXcopyorder()`.

Assigning SOS Weights

Members of an SOS should be given unique weights that in turn define the order of the variables in the set. (These unique weights are also called *reference row values*.) The most flexible way for you to assign weights is through an SOS, MPS, or LP file. An alternative is to use MPS format to assign a single reference row containing weights. Such a reference row may be a free row with specific weighting information, or it may be the objective function, or it may be a constraint row. In the Callable Library, the routine `CPXcopysos()` lets you specify weights directly in an application.

In our SOS example, page 244, we used the coefficients of the warehouse capacity constraint to assign weights.

Using Semi-Continuous Variables

Semi-continuous variables are variables that may take either the value `0` or values in a finite range [*a*, *b*]. Semi-continuous variables can be specified in MPS and LP files. In the Concert Technology Library, semi-continuous variables are instances of the class `IloSemiContVar`. In the Callable Library, semi-continuous variables can be entered with type `CPX_SEMICONT` or `CPX_SEMIINT` via the routine `CPXcopyctype()`.

Progress Reports: Interpreting the Node Log

As we explained earlier, when ILOG CPLEX optimizes mixed integer programs, it builds a tree with the linear relaxation of the original MIP at the root and subproblems to optimize at the nodes of the tree. ILOG CPLEX reports its progress in optimizing the original problem in a *node log file* as it traverses this tree.

You control how information in the log file is recorded and displayed, through two ILOG CPLEX parameters. The `MIPDisplay` parameter controls the general nature of the output that goes to the node log, as follows:

Table 8.11 *Values of the MIP Display Parameter*

Value	Effect
`0`	no display
`1`	display integer feasible solutions
`2`	display nodes under mip interval control
`3`	same as 2, but add information on node cuts

Table 8.11 *Values of the MIP Display Parameter*

Value	Effect
4	same as 3, but add LP display for root node
5	same as 3, but add LP display for all nodes

The `MIPInterval` parameter controls how frequently node log lines are printed. Its default is `100` and can be set to any positive integer value. A line is recorded in the node log for every node with an integer solution if the `MIPDisplay` parameter is set to `1` or higher, and also for any node whose number is a multiple of the `MIPInterval` value if the `MIPDisplay` is set to `2` or higher.

Here is an example of a log file from the Interactive Optimizer, where the `MIPInterval` parameter has been set to 10:

```
Tried aggregator 1 time.
No MIP presolve or aggregator reductions.
Presolve time =    0.00 sec.
Root relaxation solution time = 0.00 sec
Objective is integral.

     Nodes                                         Cuts/
  Node Left   Objective  IInf  Best Integer    Best Node  ItCnt Gap

     0   0     4.0000     6                      4.0000     12
*    4   2     5.0000     0        5.0000        4.0000     17 20.00%
    10   1     cutoff             5.0000        4.0000     31 20.00%

Integer optimal solution:  Objective =   5.0000000000e+000
Solution time =    0.02 sec.    Iterations = 41   Nodes = 13
```

In that example, ILOG CPLEX found the optimal objective function value of 5.0 after exploring 13 nodes and performing 41 (dual simplex) iterations, and ILOG CPLEX found an optimal integer solution at node 4. The MIP interval parameter was set at 10, so every tenth node was logged, in addition to the node where an integer solution was found.

As you can see in that example, ILOG CPLEX logs an asterisk (`*`) in the left-most column for any node where it finds an integer-feasible solution. In the next column, it logs the *node number*. It next logs the number of nodes left to explore.

In the next column, ILOG CPLEX either records the *objective value* at the node or a *reason to fathom* the node. (A node is fathomed if the solution of a subproblem at the node is infeasible; or if the value of objective function at the node is worse than the cutoff value for branch & cut; or if the node supplies an integer solution.)

In the column labeled `IInf`, ILOG CPLEX records the number of integer-infeasible variables and special ordered sets. If no solution has been found, the next column is left blank; otherwise, it records the best integer solution found so far.

The column labeled `Cuts/Best Node` records the best objective function value of all the unexplored nodes. If the word `Cuts` appears in this column, it means various cuts were generated; if a particular name of a cut appears, then only that kind of cut was generated.

The column labeled `ItCnt` records the cumulative iteration count of the algorithm solving the subproblems. Until a solution has been found, the column labeled `Gap` is blank. If a solution has been found, the relative gap value is printed when it is less than `999.99`; otherwise, hyphens are printed. The gap is computed as `abs(best integer - best node)/(1e-10 + abs(best integer))`. Consequently, the printed gap value may not always move smoothly. In particular, there may be sharp improvements whenever a new best integer solution is found.

ILOG CPLEX also logs its addition of cuts to a model. Here is an example of a node log file from a problem where ILOG CPLEX added several cover cuts.

```
MIP Presolve eliminated 0 rows and 1 columns.
MIP Presolve modified 12 coefficients.
Reduced MIP has 15 rows, 32 columns, and 97 nonzeros.
Presolve time =    0.00 sec.

    Nodes                                         Cuts/
 Node  Left   Objective  IInf  Best Integer  Best Node  ItCnt Gap

    0     0   2819.3574     7                2819.3574     35
              2881.8340     8                Covers: 4     44
              2881.8340    12                Covers: 3     48
*   7     6   3089.0000     0     3089.0000  2904.0815     62 5.99%

Cover cuts applied:  30

Integer optimal solution:  Objective =   3.0890000000e+003
Solution time =    0.10 sec.    Iterations = 192   Nodes = 44
```

ILOG CPLEX also logs the number of clique inequalities in the clique table at the beginning of optimization. Cuts generated at intermediate nodes are not logged individually unless they happen to be generated at a node logged for other reasons. ILOG CPLEX logs the number of applied cuts of all classes at the end.

ILOG CPLEX also indicates, in the node log file, each instance of a successful application of the node heuristic. The following example shows a node log file for a problem where the heuristic found a solution at node 0. The + denotes a node generated by the heuristic.

```
 Nodes                                         Cuts/
  Node  Left   Objective  IInf  Best Integer  Best Node  ItCnt Gap
    0     0      403.8465  640                403.8465   4037
                405.2839  609              Cliques: 10   5208
                405.2891  612              Cliques:  2   5288
Heuristic: feasible at 437.000, still looking
Heuristic: feasible at 437.000, still looking
Heuristic complete
*   0+    0      436.0000    0      436.0000   405.2891   5288 7.04%
```

Periodically, if the MIP display parameter is 2 or greater, ILOG CPLEX records the cumulative time spent since the beginning of the current MIP optimization and the amount of memory used by branch & cut. (By periodically, we mean that time and memory information appears either every 20 nodes or ten times the MIP display parameter, whichever is greater.) The following example shows you one line from a node log file indicating elapsed time and memory use.

```
Elapsed b&c time = 120.01 sec. (tree size =  0.09 MB)
```

The Interactive Optimizer prints an additional summary line in the log if optimization stops before it is complete. This summary line shows the best MIP bound, that is, the best objective value among all the remaining node subproblems. The following example shows you lines from a node log file where an integer solution has not yet been found, and the best remaining objective value is 2973.9912281.

```
Node limit, no integer solution.
Current MIP best bound =    2.9739912281e+03 (gap is infinite)
Solution time =    0.01 sec.  Iterations = 68  Nodes = 7 (7)
```

Sample: Stating a MIP Problem on page 224 presents a typical MIP problem. Here is the node log file for that problem with the default setting of the MIP display parameter:

```
Tried aggregator 1 time.
Aggregator did 1 substitutions.
Reduced MIP has 2 rows, 3 columns, and 6 nonzeros.
Presolve time =    0.00 sec.
Clique table:0 GUB, 0 GUBEQ, 0 two-covers, 0 probed
ImplBd table: 0 bounds
Root relaxation solution time =    0.00 sec.

      Nodes                                         Cuts/
 Node  Left  Objective  IInf  Best Integer  Best Node  ItCnt Gap

   0     0    125.2083     1                125.2083      3
*              122.5000     0      122.5000   Cuts:  2      4

Mixed integer rounding cuts applied: 1

Integer optimal solution:  Objective =   1.2250000000e+002
Solution time =    0.02 sec.    Iterations = 4   Nodes = 0
```

These additional items appear only in the node log file (not on screen):

- `Variable` records the name of the variable where ILOG CPLEX branched to create this node. If the branch was due to a special ordered set, the name listed here will be the right-most variable in the left subset.
- `B` indicates the branching direction:
  - `D` means the variables was restricted to a lower value;
  - `U` means the variable was restricted to a higher value;

  - `L` means the left subset of the special ordered set was restricted to 0 (zero);
  - `R` means the right subset of the special ordered set was restricted to 0 (zero).
- `Parent` indicates the node number of the parent.
- `Depth` indicates the depth of this node in the branch & cut tree.

Troubleshooting MIP Performance Problems

Even the most sophisticated methods currently available to solve pure integer and mixed integer programming problems require noticeably more computation than the methods for similarly sized continuous problems. Many relatively small integer programming models still take enormous amounts of computing time to solve. Indeed, some such models have never yet been solved. In the face of these practical obstacles to a solution, proper formulation of the model is crucial to successful solution of pure integer or mixed integer programs.

For help in formulating a model of your own integer or mixed integer problem, you may want to consult H.P. Williams's textbook about practical model building (referenced in *Further Reading* on page 30 in this manual).

Also you may want to develop a better understanding of the branch & cut algorithm. For that purpose, Williams's book offers a good introduction, and Nemhauser and Wolsey's book (also referenced in *Further Reading* on page 30 in this manual) goes into greater depth about branch & cut as well as other techniques implemented in the ILOG CPLEX MIP Optimizer.

The section *Performance Features of the Mixed Integer Optimizer* in this chapter has already discussed several specific features that are important for performance tuning of difficult models. Here, we turn attention to specific performance symptoms and the remedies that can be tried.

Too Much Time at Node 0

If you observe that a very long time passes before the branch & cut algorithm begins processing nodes, it may be that the root relaxation problem itself is taking a long time. The standard screen display will print a line telling "`Root relaxation solution time =`" once this root solve is complete, and a large solution time would be an indicator of an opportunity for tuning. If you set the `MIPDisplay` parameter to `4`, you may get a further indication of the difficulties this root solve has run into. Tuning techniques found in Chapter 5, *Solving Linear Programming Problems* or Chapter 7, *Solving Quadratic Programming Problems*, would be applicable to tuning the root solve of a MIP model too. In particular, it is worth considering setting the `RootAlg` parameter to a nondefault setting, such as the Barrier optimizer, to see if a simple change in algorithm will speed up this step sufficiently.

For some problems, ILOG CPLEX will spend a significant amount of time performing computation at node 0, apart from solving the LP or QP relaxation. While this investment of time normally saves in the overall branch & cut, it does not always do so. Time spent at node 0 can be reduced by two parameters.

First, you can try turning off the node heuristic by setting the parameter `HeurFreq` to `-1`. Second, try a less expensive variable selection strategy by setting the parameter `VarSel` to `4`, pseudo reduced costs.

It is worth noting that setting the `MIPEmphasis` parameter to `1`, resulting in an emphasis on feasibility instead of optimality, often also speeds up the processing of the root node. If your purposes are compatible with this emphasis, consider using it.

Trouble Finding More than One Feasible Solution

For some models, ILOG CPLEX finds an integer feasible solution early in the process and then does not find a better one for quite a while. One possibility, of course, is that the first feasible solution is optimal. In that case, there are no better solutions.

One possible approach to finding more feasible solutions is to increase the frequency of the node heuristic, by setting the `HeurFreq` parameter to a value like `10` or `5` or even `1`. This heuristic can be expensive, so exercise caution when setting this parameter to values less than `10`.

Another approach to finding more feasible solutions is to try a node selection strategy alternative. Setting the `NodeSel` parameter to `2` invokes a best-estimate search, which sometimes does a better job of locating good quality feasible solutions than the default node selection strategy.

Large Number of Unhelpful Cuts

While the cuts added by ILOG CPLEX reduce runtime for most problems, on occasion they can have the opposite effect. If you notice, for example, that ILOG CPLEX adds a large number of cuts at the root, but the objective value does not change significantly, then you may want to experiment with turning off cuts.

- In the Interactive Optimizer, you can turn cuts off selectively (`set mip cuts covers -1`) or all at once (`set mip cuts all -1`).
- In the Component Libraries, set the parameters that control classes of cuts The parameters are described in Table 8.7 on page 237.

Lack of Movement in the Best Node

For some models, the `Best Node` value in the node log changes very slowly or not at all. Runtimes for such models can sometimes be reduced by the variable selection strategy

known as *strong branching*. Strong branching explores a set of candidate branching-variables in-depth, performing a limited number of simplex iterations to estimate the effect of branching up or down on each.

> ***Important:*** *Strong branching consumes significantly more computation time per node than the default variable selection strategy.*

To activate strong branching, set the `VarSel` parameter to a value of `3`.

On rare occasions, it can be helpful to modify strong branching limits. If you modify the limit on the size of the candidate list, then strong branching will explore a larger (or smaller) set of candidates. If you modify the limit on strong branching iteration, then strong branching will perform more (or fewer) simplex iterations per candidate.

These limits are controlled by the parameters `StrongCandLim` and `StrongItLim`, respectively.

Other parameters to consider trying, in the case of slow movement of the Best Node value, are non-default levels for `Probe` (try the aggressive setting of `3` first, and then reduce it if the probing step itself takes excessive time for your purposes), and `MIPEmphasis` set to a value of `3`.

Time Wasted on Overly Tight Optimality Criteria

Sometimes ILOG CPLEX finds a good integer solution early, but must examine many additional nodes to prove the solution is optimal. You can speed up the process in such a case if you are willing to change the optimality tolerance. ILOG CPLEX supports two kinds of tolerance:

- *Relative optimality tolerance* guarantees that a solution lies within a certain percentage of the optimal solution.
- *Absolute optimality tolerance* guarantees that a solution lies within a certain absolute range of the optimal solution.

The default relative optimality tolerance is 0.0001. At this tolerance, the final integer solution is guaranteed to be within 0.01% of the optimal value. Of course, many formulations of integer or mixed integer programs do not require such tight tolerance, so requiring ILOG CPLEX to seek integer solutions that meet this tolerance in those cases is wasted computation. If you can accept greater optimality tolerance in your model, then you should change the parameter `EpGap`.

If, however, you know that the objective values of your problem are near zero, then you should change the absolute gap because percentages of very small numbers are less useful as optimality tolerance. Change the parameter `EpAGap` in this case.

To speed up the proof of optimality, you can set objective difference parameters, both relative and absolute. Setting these parameters helps when there are many integer solutions

with similar objective values. For example, setting the `ObjDif` parameter to `100.0` makes ILOG CPLEX skip any potential solution with its objective value within 100.0 units of the best integer solution so far. Or, setting the `RelObjDif` to `0.01` would mean that ILOG CPLEX would skip any potential new solution that is not at least 1% better than the incumbent solution. Naturally, since this objective difference setting may make ILOG CPLEX skip an interval where the true integer optimum may be found, the objective difference setting weakens the guarantee of optimality.

Cutoff parameters can also be helpful in restricting the search for optimality. If you know that there are solutions within a certain distance of the initial relaxation of your problem, then you can readily set the upper cutoff parameter for minimization problems and the lower cutoff parameter for maximization problems. Set the parameters `CutUp` and `CutLo`, respectively, to establish a cutoff value.

When you set a MIP cutoff value, ILOG CPLEX searches with the same solution strategy as though it had already found an integer solution, using a node selection strategy that differs from the one it uses before a first solution has been found.

Running Out of Memory

A very common difficulty with MIPs is running out of memory. This problem occurs when the branch & cut tree becomes so large that insufficient memory remains to solve an LP or QP subproblem. As memory gets tight, you may observe warning messages from ILOG CPLEX as it attempts various operations in spite of limited memory. In such a situation, if ILOG CPLEX does not find a solution shortly, it terminates the process with an error message.

The information about a tree that ILOG CPLEX accumulates in memory can be substantial. In particular, ILOG CPLEX saves a basis for every unexplored node. Furthermore, when ILOG CPLEX uses the best bound or best estimate strategies of node selection, the list of unexplored nodes itself can become very long for large or difficult problems. How large the unexplored node list can be depends on the actual amount of memory available, the size of the problem, and algorithm selected.

A less frequent cause of memory consumption is the generation of cutting planes. Gomory fractional cuts, and, in rare instances, Mixed Integer Rounding cuts, are the ones most likely to be dense and thus use significant memory under default/automatic settings. You can try turning off these cuts, or any of the cuts you see listed as being generated for your model (in the cuts summary at the end of the node log), or simply all cuts, through the use of parameter settings discussed in the section on cuts in this manual; doing this carries the risk that this will make the model harder to solve and only delay the eventual exhaustion of available memory during branching.

Certainly, if you increase the amount of available memory, you extend the problem-solving capability of ILOG CPLEX. Unfortunately, when a problem fails because of insufficient memory, it is difficult to project how much further the process needed to go and how much

more memory is needed to solve the problem. For these reasons, the following suggestions aim at avoiding memory failure whenever possible and recovering gracefully otherwise.

Reset the Tree Memory Parameter

To avoid a failure due to running out of memory, we recommend setting the working memory parameter, `WorkMem`, to a value significantly lower than the available memory on your computer (in megabytes), to instruct ILOG CPLEX to begin compressing the storage of nodes before it consumes all of available memory. See the related topic *Use Node Files for Storage* on page 254, for other choices of what should happen when `WorkMem` is exceeded.

Because the storage of nodes can require a lot of space, it may also be advisable to set a tree limit on the size of the entire tree being stored so that not all of your disk will be filled up with working storage. The call to the MIP optimizer will be stopped once the size of the tree exceeds the value of `TreLim`, the tree limit parameter. Under default settings the limit is infinity ($1e^{+75}$), but you can set it to a lower value (in megabytes)

Write a Tree File and Restart

On some platforms, even when the current tree size is within system limits, memory fragmentation may be so great that performance becomes poor. To overcome that kind of fragmentation, we recommend that you stop optimization, write a tree file (using the TRE format), exit ILOG CPLEX, restart it, read in the model and tree file, and continue optimization then.

Use Node Files for Storage

ILOG CPLEX offers a node file storage feature to store some parts of the branch & cut tree in files. If you use this feature, ILOG CPLEX will be able to explore more nodes within a smaller amount of computer memory.This feature includes several options to reduce the use of physical memory, and it entails a very small increase in runtime. Node file storage offers a much better option than relying on swap space.

This feature is especially helpful when you are using steepest-edge pricing as the subproblem simplex pricing strategy because pricing information itself consumes a great deal of memory.

There are several parameters that control the use of node files. They are:

- `WorkMem`
- `NodeFileInd`
- `TreLim`
- `WorkDir`

When tree storage size exceeds the limit defined by `WorkMem`, what happens next is determined by the setting of `NodeFileInd`. If the latter parameter is set to zero, then optimization proceeds with the tree stored in memory until ILOG CPLEX reaches the tree memory limit (`TreLim`). If the `NodeFileInd` parameter is set to `1` (the default), then a fast

compression algorithm is used on the nodes to try to conserve memory, without resorting to writing the node files to disk. If the parameter is set to 2, then node files are written to disk. If the parameter is set to 3, then nodes are both compressed (as in option 1) and written to disk (as in option 2). Thus, regardless of the setting of `NodeFileInd`, ILOG CPLEX will stop the optimization when the total memory used to store the tree exceeds the tree memory limit Table 8.12 summarizes these different options.

Table 8.12 *Values for the Node File Storage Parameter*

Value	Meaning	Comments
`0`	no node files	optimization continues
`1`	node file in memory and compressed	optimization continues (default)
`2`	node file on disk	files created in temporary directory
`3`	node file on disk and compressed	files created in temporary directory

In cases where node files are written to disk, ILOG CPLEX will create a temporary subdirectory under the directory specified by the `WorkDir` parameter. The directory named by this parameter must exist before ILOG CPLEX attempts to create node files. By default, the value of this parameter is ".", which means the current working directory.

ILOG CPLEX creates the temporary directory by means of system calls. If the system environment variable is set (on Windows platforms, the environment variable `TMP`; on UNIX platforms, the environment variable `TMPDIR`), then the system ignores the ILOG CPLEX node-file directory parameter and creates the temporary node-file directory in the location indicated by its system environment variable. Furthermore, if the directory specified in the ILOG CPLEX node-file directory parameter is invalid (for example, if it contains illegal characters, or if the directory does not allow write access), then the system chooses a location according to its own logic.

The temporary directory created for node file storage will have a name prefixed by `cpx`. The files within it will also have names prefixed by `cpx`.

ILOG CPLEX automatically removes the files and their temporary directory when it frees the branch & cut tree:

- in the Interactive Optimizer,
  - at problem modification;
  - at normal termination;
- from the Concert Technology Library,
  - when you call `env.end()`
  - when you modify the extracted model

- from the Callable Library,
  - when you call a problem modification routine;
  - when you call `CPXfreeprob()`.

If a program terminates abnormally, the files are not removed.

ILOG CPLEX uses node file storage most effectively when the amount of working memory is reasonably large so that it does not have to create node files too frequently. A reasonable amount is to use approximately half the memory, but no more than 128 megabytes. Higher values result in only marginally improved efficiency.

Node files could grow very large. Use the `TreMem` parameter to limit the size of the tree so that it does not exceed available disk space, when you choose `NodeFileInd` settings `2` or `3`. It is usually better to let ILOG CPLEX terminate the run gracefully, with whatever current feasible solution has been found, than to trigger an error message or even a program abort.

When ILOG CPLEX uses node-file storage, the sequence of nodes processed may differ from the sequence in which nodes are processed without node-file storage. Nodes in node-file storage are not accessible to user-written callback routines.

Change Algorithms

The best approach to reduce memory use is to modify the solution process. Here are some ways to do so:

- Switch the node selection strategy to best estimate, or more drastically to depth-first, as explained on page 237. Depth-first search rarely generates a long, memory-consuming list of unexplored nodes since ILOG CPLEX dives deeply into the tree instead of jumping around. A narrowly focused search, like depth-first, also often results in faster processing times for individual nodes. However, overall solution time is generally much worse than with best-bound node selection because each branch is searched exhaustively to its deepest level before it is fathomed in favor of better branches.
- Another memory-conserving strategy is to use strong branching (`VarSel` parameter value `3`) for variable selection. Strong branching requires substantial computational effort at each node to determine the best branching variable. As a result, it generates fewer nodes and thus makes less overall demand on memory. Often, strong branching is faster as well.

Difficulty Solving Subproblems

There are classes of MIPs that produce very difficult subproblems, for example, if the subproblems are dual degenerate. In such a case, a different optimizer, such as the primal simplex or the barrier optimizer, may be better suited to your problem than the default dual simplex optimizer for subproblems. These alternatives are discussed in *Subproblem Optimization* on page 257. A stronger dual pricing algorithm, such as dual steepest-edge pricing (the parameter `DPriInd` set to `2`), could also be considered.

Overcoming Degeneracy

If the subproblems are dual degenerate, then consider using the primal simplex optimizer for the subproblems. You make this change by setting the `SubAlg` parameter to `1`.

Another effective strategy in overcoming dual degeneracy is to permanently perturb the problem. For subproblems that are dual degenerate, in the Interactive Optimizer, write out the perturbed problem as a DPE file with the command `write` *`filename`*`.dpe` substituting an appropriate file name. (A `.dpe` file is saved as a binary SAV format file.) Then you can read the saved file back in and solve it. The subproblem should then solve more cleanly and quickly.

In the case of DPE files solved by the dual simplex optimizer, any integer solution is also guaranteed to be an integer-feasible solution to the original problem. In most cases, the solution will be optimal or near-optimal as well.

Subproblem Optimization

In some problems, you can improve performance by evaluating how the LP or QP subproblems are solved at the nodes in the branch & cut tree, and then possibly modifying the choice of algorithm to solve them. You can control which algorithm ILOG CPLEX applies to the initial relaxation of your problem separately from your control of which algorithm ILOG CPLEX applies to other subproblems. The following sections explain those parameters more fully.

RootAlg Parameter

The `RootAlg` algorithm parameter indicates the algorithm for ILOG CPLEX to use on the *initial* subproblem. In a typical MIP, that initial subproblem is usually the linear relaxation of the original MIP. By default, ILOG CPLEX starts the initial subproblem with the dual simplex optimizer. You may have information about your problem that indicates another optimizer could be more efficient. Table 8.13 summarizes the values available for the `RootAlg` parameter.

To set this parameter:

- In the Interactive Optimizer, use the command `set mip strategy startalgorithm` with the value to indicate the optimizer you want.
- In the Concert Technology library, use the method `IloCplex` method `setParam()` with the parameter `RootAlg` and the appropriate algorithm enumeration value.
- In the Callable Library, use the routine `CPXsetintparam()` with the parameter `CPX_PARAM_STARTALG`, and the appropriate symbolic constant.

Table 8.13 *Values of RootAlg and NodeAlg Parameters*

Concert Technology Library Enumeration	Callable Library Symbolic Constant	Value	Calls this Optimizer
`Auto`	`CPX_ALG_AUTO`	0	automatic
`Primal`	`CPX_ALG_PRIMAL`	1	primal simplex
`Dual`	`CPX_ALG_DUAL`	2	dual simplex (default)
`Network`	`CPX_ALG_HYBNETOPT`	3	network simplex
`Barrier`	`CPX_ALG_BARRIER`	4	barrier with crossover
`Sifting`	`CPX_ALG_SIFTING`	5	sifting

NodeAlg Parameter

The `NodeAlg` parameter indicates the algorithm for ILOG CPLEX to use on node relaxations other than the root node. By default, ILOG CPLEX applies the dual simplex optimizer to subproblems, and unlike the `RootAlg` parameter it is extremely unusual for this to not be the most desirable choice, but again, you may have information about your problem that tells you another optimizer could be more efficient. The values and symbolic constants are the same for the `NodeAlg` parameter as for the `RootAlg` parameter in Table 8.13.

To set the `NodeAlg` parameter:

- In the Concert Technology library, use the `IloCplex` method `setParam()` with the parameter `NodeAlg` and the appropriate algorithm enumeration value.
- In the Callable Library, use the routine `CPXsetintparam()` with the parameter `CPX_PARAM_SUBALG`, and the appropriate symbolic constant.
- In the Interactive Optimizer, use the command "`set mip strategy subalgorithm`" with the value to indicate the optimizer you want.

Example: Optimizing a Basic MIP Problem

This example illustrates how to optimize a MIP with the ILOG CPLEX Component Libraries.

Complete Program: ilomipex1.cpp

The example derives from `ilolpex8.cpp`. Here are the differences between that linear program and this mixed integer program:

- The problem to solve is slightly different. It appears in *Sample: Stating a MIP Problem* on page 224.
- The routine `populatebyrow()` added the variables, objective, and constraints to the model created by the method `IloModel model(env)`.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilomipex1.cpp
// Version 8.1
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilomipex1.cpp - Entering and optimizing a MIP problem

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
      IloRangeArray con(env);
      populatebyrow (model, var, con);

      IloCplex cplex(model);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;
```

```
      cplex.exportModel("mipex1.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]             <= 30);
   c.add(              x[1]        - 3.5* x[3] == 0);
   model.add(c);

}  // END populatebyrow
```

Complete Program: mipex1.c

The example derives from `lpex8.c`. Here are the differences between that linear program and this mixed integer program:

- The problem to solve is slightly different. It appears in *Sample: Stating a MIP Problem* on page 224.
- The routine `setproblemdata()` has a parameter, `ctype`, to set the types of the variables to indicate which ones must assume integer values. The routine `CPXcopyctype()` associates this data with the problem that `CPXcreateprob()` creates.
- The example calls `CPXmipopt()` to optimize the problem, rather than `CPXlpopt()`.

- The example calls the routines `CPXgetstat()`, `CPXgetmipobjval()`, `CPXgetmipx()`, and `CPXgetmipslack()` (instead of `CPXsolution()`) to get a solution.

  We do not get dual variables this way. If we want dual variables, we must do the following:

  - Use `CPXchgprobtype()` to change the problem type to `CPXPROB_FIXEDMILP`.
  - Then call `CPXprimopt()` to optimize that problem.
  - Then use `CPXsolution()` to get a solution to the *fixed* problem.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex1.c                                             */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* mipex1.c - Entering and optimizing a MIP problem */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the include of cplex.h. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>
#include <stdlib.h>

/* Include declaration for function at end of program */

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p);

static void
   free_and_null (char **ptr);


/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    3
#define NUMCOLS    4
```

```
#define NUMNZ      9


int
main (void)
{
/* Declare pointers for the variables and arrays that will contain
   the data which define the LP problem.  The setproblemdata() routine
   allocates space for the problem data.  */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
   int      *matcnt = NULL;
   int      *matind = NULL;
   double   *matval = NULL;
   double   *lb = NULL;
   double   *ub = NULL;
   char     *ctype = NULL;

   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      variable values, and row slacks. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   slack[NUMROWS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
```

```
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Fill in the data for the problem.  */

status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                         &lb, &ub, &ctype);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now copy the problem data into the lp */

status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                    sense, matbeg, matcnt, matind, matval,
                    lb, ub, NULL);

if ( status ) {
   fprintf (stderr, "Failed to copy problem data.\n");
   goto TERMINATE;
}
```

```
/* Now copy the ctype array */

status = CPXcopyctype (env, lp, ctype);
if ( status ) {
   fprintf (stderr, "Failed to copy ctype\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);

/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);

status = CPXgetmipobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"No MIP objective value available.  Exiting...\n");
   goto TERMINATE;
}

printf ("Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using what was passed to CPXcopylp.
   cur_numrows and cur_numcols store the current number of rows and
   columns, respectively.  */

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal integer x.\n");
   goto TERMINATE;
}

status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal slack values.\n");
   goto TERMINATE;
}

for (i = 0; i < cur_numrows; i++) {
```

```
      printf ("Row %d:  Slack = %10f\n", i, slack[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f\n", j, x[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "mipex1.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
```

```
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x4
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   char     *zctype = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
   zsense    = (char *) malloc (NUMROWS * sizeof(char));
```

```
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zctype    = (char *) malloc (NUMCOLS * sizeof(char));

if ( zprobname == NULL || zobj    == NULL ||
     zrhs      == NULL || zsense  == NULL ||
     zmatbeg   == NULL || zmatcnt == NULL ||
     zmatind   == NULL || zmatval == NULL ||
     zlb       == NULL || zub     == NULL ||
     zctype    == NULL                       )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0]  = 1.0;   zobj[1]   = 2.0;  zobj[2]    = 3.0;    zobj[3] = 1.0;

zmatbeg[0] = 0;    zmatbeg[1] = 2;    zmatbeg[2] = 5;   zmatbeg[3] = 7;
zmatcnt[0] = 2;    zmatcnt[1] = 3;    zmatcnt[2] = 2;   zmatcnt[3] = 2;

zmatind[0] = 0;    zmatind[2] = 0;    zmatind[5] = 0;   zmatind[7] = 0;
zmatval[0] = -1.0; zmatval[2] = 1.0;  zmatval[5] = 1.0; zmatval[7] = 10.0;

zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                   zmatind[4] = 2;                     zmatind[8] = 2;
                   zmatval[4] = 1.0;                   zmatval[8] = -3.5;

zlb[0] = 0.0;   zlb[1] = 0.0;      zlb[2] = 0.0;     zlb[3] = 2.0;
zub[0] = 40.0;  zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

 zctype[0] = 'C';    zctype[1] = 'C';   zctype[2] = 'C';   zctype[3] = 'I';

/* The right-hand-side values don't fit nicely on a line above.  So put
   them here.  */

zsense[0] = 'L';
zrhs[0]   = 20.0;

zsense[1] = 'L';
zrhs[1]   = 30.0;
```

```
   zsense[2] = 'E';
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);

}  /* END setproblemdata */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

Example: Reading a MIP Problem from a File

This example shows you how to solve a MIP with the Component Libraries when the problem data is stored in a file.

Complete Program: ilomipex2.cpp

This example derives from `ilolpex2.cpp`, an LP example explained in the manual ILOG CPLEX *Getting Started* manual. That LP example differs from this MIP example in these ways:

- This example solves only MIPs, so it calls only `IloCplex::solve()`, and its command line does not require the user to indicate an optimizer.
- This example does not generate or print a basis.

Like other applications based on the ILOG CPLEX Concert Technology Library, this one uses `IloEnv env` to initialize the Concert Technology environment and `IloModel model(env)` to create a problem object. Before it ends, it calls `env.end()` to free the environment.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilomipex2.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// ilomipex2.cpp - Reading in and optimizing a problem
//
// To run this example, command line arguments are required.
// i.e.,   ilomipex2   filename
//
// Example:
//     ilomipex2  mexample.mps
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

static void usage (const char *progname);

int
main (int argc, char **argv)
{
   IloEnv env;
```

```
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
} // END usage
```

Complete Program: mipex2.c

The example derives from `lpex2.c`, an LP example explained in the ILOG CPLEX *Getting Started* manual. That LP example differs from this MIP example in these ways:

- This example solves only MIPs, so it calls only `CPXmipopt()`, and its command line does not require the user to indicate an optimizer.
- This example calls `CPXgetstat()`, `CPXgetmipobjval()`, and `CPXgetmipx()` to get a solution. It doesn't generate or print a basis.

Like other applications based on the ILOG CPLEX Callable Library, this one calls `CPXopenCPLEX()` to initialize the ILOG CPLEX environment; it sets the screen-indicator parameter to direct output to the screen and calls `CPXcreateprob()` to create a problem object. Before it ends, it calls `CPXfreeprob()` to free the space allocated to the problem object and `CPXcloseCPLEX()` to free the environment.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex2.c                                             */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* mipex2.c - Reading in and optimizing a MIP problem */

/* To run this example, command line arguments are required.
   i.e.,   mipex2   filename
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
   Example:
       mipex2  mexample.mps
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

static void
   free_and_null (char **ptr),
   usage         (char *progname);
```

```
int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      and variable values. */


   int      solstat;
   double   objval;
   double   *x     = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           j;
   int           cur_numcols;

   /* Check the command line arguments */

   if ( argc != 2 ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */

   env = CPXopenCPLEX (&status);

   /* If an error occurs, the status value indicates the reason for
      failure.  A call to CPXgeterrorstring will produce the text of
      the error message.  Note that CPXopenCPLEX produces no output,
      so the only way to see the cause of the error is to use
      CPXgeterrorstring.  For other CPLEX routines, the errors will
      be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

   if ( env == NULL ) {
      char  errmsg[1024];
      fprintf (stderr, "Could not open CPLEX environment.\n");
      CPXgeterrorstring (env, status, errmsg);
      fprintf (stderr, "%s", errmsg);
      goto TERMINATE;
   }

   /* Turn on output to the screen */

   status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
   if ( status ) {
      fprintf (stderr,
               "Failure to turn on screen indicator, error %d.\n", status);
      goto TERMINATE;
   }

   /* Create the problem, using the filename as the problem name */

   lp = CPXcreateprob (env, &status, argv[1]);
```

```
/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);
printf ("Solution status %d.\n", solstat);

status  = CPXgetmipobjval (env, lp, &objval);

if ( status ) {
   fprintf (stderr,"Failed to obtain objective value.\n");
   goto TERMINATE;
}

printf ("Objective value %.10g\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is. cur_numcols stores the current number
   of columns. */

cur_numcols = CPXgetnumcols (env, lp);

/* Allocate space for solution */

x = (double *) malloc (cur_numcols*sizeof(double));

if ( x == NULL ) {
   fprintf (stderr, "No memory for solution values.\n");
   goto TERMINATE;
}
```

```
   status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, "Failed to obtain solution.\n");
      goto TERMINATE;
   }

   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      printf ( "Column %d:  Value = %17.10g\n", j, x[j]);
   }



TERMINATE:

   /* Free up the solution */

   free_and_null ((char **) &x);

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
      char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
```

```
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */


static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"  This program uses the CPLEX MIP optimizer.\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Example: Using SOS and Priority

This example illustrates how to use SOS and priority orders.

Complete Program: ilomipex3.cpp

It derives from `ilomipex1.cpp`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `setPriorities()` sets the SOS and priority order:

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilomipex3.cpp
// Version 8.1
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilomipex3.cpp - Entering and optimizing a MIP problem with SOS sets
//                 and priority orders.  Is a modification of ilomipex1.cpp.
//                 Note that the problem solved is slightly different than
//                 mipex1.c so that the output is interesting.

#include <ilcplex/ilocplex.h>
```

```
ILOSTLBEGIN

static void
   populatebyrow(IloModel model, IloNumVarArray var, IloRangeArray con);

int
main (void) {
   IloEnv env;
   try {
      IloModel model(env);

      IloNumVarArray var(env);
      IloRangeArray con(env);
      populatebyrow (model, var, con);

      IloCplex cplex(model);
      IloNumVarArray ordvar(env, 2, var[1], var[3]);
      IloNumArray    ordpri(env, 2, 8.0, 7.0);
      cplex.setPriorities (ordvar, ordpri);
      cplex.setDirection(var[1], IloCplex::BranchUp);
      cplex.setDirection(var[3], IloCplex::BranchDown);
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, con);
      env.out() << "Slacks        = " << vals << endl;

      cplex.exportModel("mipex3.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;

}  // END main


static void
populatebyrow (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   x.add(IloNumVar(env, 0.0, 40.0));
```

```
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 0.0, IloInfinity, ILOINT));
   x.add(IloNumVar(env, 2.0, 3.0, ILOINT));
   model.add(IloMaximize(env, x[0] + 2 * x[1] + 3 * x[2] + x[3]));

   c.add( - x[0] +     x[1] + x[2] + 10 * x[3] <= 20);
   c.add(   x[0] - 3 * x[1] + x[2]            <= 30);
   c.add(              x[1]        - 3.5* x[3] == 0);
   model.add(c);

   model.add(IloSOS1(model.getEnv(),
                     IloNumVarArray(model.getEnv(), 2, x[2], x[3]),
                     IloNumArray(env, 2, 25.0, 18.0)    ));

}  // END populatebyrow
```

Complete Program: mipex3.c

This example derives from `mipex1.c`. The differences between that simpler MIP example and this one are:

- The problem solved is slightly different so the output is interesting. The actual SOS and priority order that the example implements are arbitrary; they do not necessarily represent good data for this problem.
- The routine `CPXwriteprob()` writes the problem to disk before the example copies the SOS and priority order to verify that the base problem was copied correctly.
- The ILOG CPLEX preprocessing parameters for the presolver and aggregator are turned off to make the output interesting. Generally, we do not require nor recommend doing this.
- The routine `setsosandorder()` sets the SOS and priority order:
  - It calls `CPXcopysos()` to copy the SOS into the problem object.
  - It calls `CPXcopyorder()` to copy the priority order into the problem object.
  - It writes the SOS information to files by calling `CPXsoswrite()`.
  - It writes the priority order to files by calling `CPXordwrite()`.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/mipex3.c                                             */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/
```

```
/* mipex3.c - Entering and optimizing a MIP problem with SOS sets
              and priority orders.  Is a modification of mipex1.c .
              Note that the problem solved is slightly different than
              mipex1.c so that the output is interesting.  */

/* Bring in the CPLEX function declarations and the C library
   header file stdlib.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>
#include <stdlib.h>

/* Include declaration for function at end of program */

static int
   setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                   int *objsen_p, double **obj_p, double **rhs_p,
                   char **sense_p, int **matbeg_p, int **matcnt_p,
                   int **matind_p, double **matval_p,
                   double **lb_p, double **ub_p, char **ctype_p),
   setsosandorder (CPXENVptr env, CPXLPptr lp);

static void
   free_and_null (char **ptr);



/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    3
#define NUMCOLS    4
#define NUMNZ      9


int
main (void)
{
   /* Declare and allocate space for the variables and arrays that
      will contain the data which define the LP problem */

   char     *probname = NULL;
   int      numcols;
   int      numrows;
   int      objsen;
   double   *obj = NULL;
   double   *rhs = NULL;
   char     *sense = NULL;
   int      *matbeg = NULL;
```

```
int      *matcnt = NULL;
int      *matind = NULL;
double   *matval = NULL;
double   *lb = NULL;
double   *ub = NULL;
char     *ctype = NULL;

/* Declare and allocate space for the variables and arrays where we
   will store the optimization results including the status, objective
   value, variable values, and row slacks. */

int      solstat;
double   objval;
double   x[NUMCOLS];
double   slack[NUMROWS];


CPXENVptr     env = NULL;
CPXLPptr      lp = NULL;
int           status;
int           i, j;
int           cur_numrows, cur_numcols;

/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Fill in the data for the problem.  */
```

```
status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
                         &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
                         &lb, &ub, &ctype);
if ( status ) {
   fprintf (stderr, "Failed to build problem data arrays.\n");
   goto TERMINATE;
}

/* Create the problem. */

lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now copy the problem data into the lp */

status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
                    sense, matbeg, matcnt, matind, matval,
                    lb, ub, NULL);

if ( status ) {
   fprintf (stderr, "Failed to copy problem data.\n");
   goto TERMINATE;
}

/* Now copy the ctype array */

status = CPXcopyctype (env, lp, ctype);
if ( status ) {
   fprintf (stderr, "Failed to copy ctype\n");
   goto TERMINATE;
}

/* Write a copy of the problem to a file. */

status = CPXwriteprob (env, lp, "mipex3.mps", NULL);
if ( status ) {
   fprintf (stderr, "Failed to write LP to disk.\n");
   goto TERMINATE;
}

/* Set up the SOS set and priority order */
```

```
status = setsosandorder (env, lp);
if ( status ) goto TERMINATE;

/* Turn off CPLEX presolve, aggregate, and print every node.  This
   is just to make it interesting.  Turning off CPLEX presolve is
   NOT recommended practice !! */

status = CPXsetintparam (env, CPX_PARAM_PREIND, CPX_OFF);
if (!status) CPXsetintparam (env, CPX_PARAM_AGGIND, CPX_OFF);
if (!status) CPXsetintparam (env, CPX_PARAM_MIPINTERVAL, 1);

if ( status ) {
   fprintf (stderr, "Failed to set some CPLEX parameters.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXmipopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize MIP.\n");
   goto TERMINATE;
}

solstat = CPXgetstat (env, lp);

/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);

status = CPXgetmipobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr,"No MIP objective value available.  Exiting...\n");
   goto TERMINATE;
}

printf ("Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using what was passed to CPXcopylp.
   cur_numrows and cur_numcols store the current number of rows and
   columns, respectively.  */

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);

status = CPXgetmipx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to get optimal integer x.\n");
   goto TERMINATE;
```

```
   }

   status = CPXgetmipslack (env, lp, slack, 0, cur_numrows-1);
   if ( status ) {
      fprintf (stderr, "Failed to get optimal slack values.\n");
      goto TERMINATE;
   }

   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f\n", i, slack[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f\n", j, x[j]);
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   /* Free up the problem data arrays, if necessary. */

   free_and_null ((char **) &probname);
   free_and_null ((char **) &obj);
   free_and_null ((char **) &rhs);
   free_and_null ((char **) &sense);
   free_and_null ((char **) &matbeg);
```

```
   free_and_null ((char **) &matcnt);
   free_and_null ((char **) &matind);
   free_and_null ((char **) &matval);
   free_and_null ((char **) &lb);
   free_and_null ((char **) &ub);
   free_and_null ((char **) &ctype);

   return (status);

}  /* END main */


/* This function fills in the data structures for the mixed integer program:

      Maximize
       obj: x1 + 2 x2 + 3 x3 + x4
      Subject To
       c1: - x1 + x2 + x3 + 10x4  <= 20
       c2: x1 - 3 x2 + x3         <= 30
       c3:       x2       - 3.5x4  = 0
      Bounds
       0 <= x1 <= 40
       2 <= x4 <= 3
      Integers
        x2 x3 x4
      End
 */


static int
setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
                int *objsen_p, double **obj_p, double **rhs_p,
                char **sense_p, int **matbeg_p, int **matcnt_p,
                int **matind_p, double **matval_p,
                double **lb_p, double **ub_p, char **ctype_p)
{
   char     *zprobname = NULL;     /* Problem name <= 16 characters */
   double   *zobj = NULL;
   double   *zrhs = NULL;
   char     *zsense = NULL;
   int      *zmatbeg = NULL;
   int      *zmatcnt = NULL;
   int      *zmatind = NULL;
   double   *zmatval = NULL;
   double   *zlb = NULL;
   double   *zub = NULL;
   char     *zctype = NULL;
   int      status = 0;

   zprobname = (char *) malloc (16 * sizeof(char));
   zobj      = (double *) malloc (NUMCOLS * sizeof(double));
   zrhs      = (double *) malloc (NUMROWS * sizeof(double));
```

```
zsense    = (char *) malloc (NUMROWS * sizeof(char));
zmatbeg   = (int *) malloc (NUMCOLS * sizeof(int));
zmatcnt   = (int *) malloc (NUMCOLS * sizeof(int));
zmatind   = (int *) malloc (NUMNZ * sizeof(int));
zmatval   = (double *) malloc (NUMNZ * sizeof(double));
zlb       = (double *) malloc (NUMCOLS * sizeof(double));
zub       = (double *) malloc (NUMCOLS * sizeof(double));
zctype    = (char *) malloc (NUMCOLS * sizeof(char));

if ( zprobname == NULL || zobj    == NULL ||
     zrhs      == NULL || zsense  == NULL ||
     zmatbeg   == NULL || zmatcnt == NULL ||
     zmatind   == NULL || zmatval == NULL ||
     zlb       == NULL || zub     == NULL ||
     zctype    == NULL                      )  {
   status = 1;
   goto TERMINATE;
}

strcpy (zprobname, "example");

/* The code is formatted to make a visual correspondence
   between the mathematical linear program and the specific data
   items.   */

  zobj[0] = 1.0;   zobj[1]   = 2.0;  zobj[2] = 3.0;    zobj[3] = 1.0;

zmatbeg[0] = 0;    zmatbeg[1] = 2;   zmatbeg[2] = 5;   zmatbeg[3] = 7;
zmatcnt[0] = 2;    zmatcnt[1] = 3;   zmatcnt[2] = 2;   zmatcnt[3] = 2;

zmatind[0] = 0;    zmatind[2] = 0;   zmatind[5] = 0;   zmatind[7] = 0;
zmatval[0] = -1.0; zmatval[2] = 1.0; zmatval[5] = 1.0; zmatval[7] = 10.0;

zmatind[1] = 1;    zmatind[3] = 1;    zmatind[6] = 1;
zmatval[1] = 1.0;  zmatval[3] = -3.0; zmatval[6] = 1.0;

                   zmatind[4] = 2;                    zmatind[8] = 2;
                   zmatval[4] = 1.0;                  zmatval[8] = -3.5;

zlb[0] = 0.0;    zlb[1] = 0.0;          zlb[2] = 0.0;          zlb[3] = 2.0;
zub[0] = 40.0;   zub[1] = CPX_INFBOUND; zub[2] = CPX_INFBOUND; zub[3] = 3.0;

zctype[0] = 'C';   zctype[1] = 'I';  zctype[2] = 'I';   zctype[3] = 'I';

/* The right-hand-side values don't fit nicely on a line above.  So put
   them here.  */

zsense[0] = 'L';
zrhs[0]   = 20.0;

zsense[1] = 'L';
zrhs[1]   = 30.0;
```

```
   zsense[2] = 'E';
   zrhs[2]   = 0.0;

TERMINATE:

   if ( status ) {
      free_and_null ((char **) &zprobname);
      free_and_null ((char **) &zobj);
      free_and_null ((char **) &zrhs);
      free_and_null ((char **) &zsense);
      free_and_null ((char **) &zmatbeg);
      free_and_null ((char **) &zmatcnt);
      free_and_null ((char **) &zmatind);
      free_and_null ((char **) &zmatval);
      free_and_null ((char **) &zlb);
      free_and_null ((char **) &zub);
      free_and_null ((char **) &zctype);
   }
   else {
      *numcols_p   = NUMCOLS;
      *numrows_p   = NUMROWS;
      *objsen_p    = CPX_MAX;   /* The problem is maximization */

      *probname_p  = zprobname;
      *obj_p       = zobj;
      *rhs_p       = zrhs;
      *sense_p     = zsense;
      *matbeg_p    = zmatbeg;
      *matcnt_p    = zmatcnt;
      *matind_p    = zmatind;
      *matval_p    = zmatval;
      *lb_p        = zlb;
      *ub_p        = zub;
      *ctype_p     = zctype;
   }
   return (status);

}  /* END setproblemdata */


static int
setsosandorder (CPXENVptr env, CPXLPptr lp)
{
   /* Priority order information */
   int  colindex[2];
   int  priority[2];
   int  direction[2];

   /* SOS set information */
   char   sostype[1];
   int    sospri[1];
   int    sosbeg[1];
```

```
int    sosind[2];
double sosref[2];

int  status = 0;

/* Note - for this example, the priority order and SOS information
   are just made up for illustrative purposes.  The priority order
   is not necessarily a good one for this particular problem.  */

/* Set order info.  Variables 1 and 3 will be in the priority order,
   with respective priorities of 8 and 7, and with respective
   branching directions of up and down */

colindex[0]  = 1;                colindex[1]  = 3;
priority[0]  = 8;                priority[1]  = 7;
direction[0] = CPX_BRANCH_UP;    direction[1] = CPX_BRANCH_DOWN;

status = CPXcopyorder (env, lp, 2, colindex, priority, direction);
if ( status ) {
   fprintf (stderr, "CPXcopyorder failed.\n");
   goto TERMINATE;
}

/* Set SOS set info.  Create one SOS type 1 set, with variables
   2 and 3 in it, with set priority 3, and reference values
   25 and 18  for the 2 variables, respectively.   */

sostype[0] = CPX_TYPE_SOS1;
sospri[0]  = 3;
sosbeg[0]  = 0;
sosind[0]  = 2;    sosind[1] = 3;
sosref[0]  = 25;   sosref[1] = 18;

status = CPXcopysos (env, lp, 1, 2, sostype, sospri,
                     sosbeg, sosind, sosref);
if ( status ) {
   fprintf (stderr, "CPXcopysos failed.\n");
   goto TERMINATE;
}

/* To assist in debugging, write the order and SOS to a file. */

status = CPXordwrite (env, lp, "mipex3.ord");
if ( status ) {
   fprintf (stderr, "CPXordwrite failed.\n");
   goto TERMINATE;
}

status = CPXsoswrite (env, lp, "mipex3.sos");
if ( status ) {
   fprintf (stderr, "CPXsoswrite failed.\n");
   goto TERMINATE;
}
```

```
TERMINATE:

   return (status);

}



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

C H A P T E R

9

More About Using ILOG CPLEX

This chapter provides information designed to help you master several important aspects of ILOG CPLEX. It includes sections on:

- Managing Input & Output
- Using Query Routines in the Callable Library
- Goals in IloCplex
- Using Callbacks
- A Comparison of Goals and Callbacks
- Using Parallel Optimizers

Managing Input & Output

This section tells you about input to and output from ILOG CPLEX. It contains the following subsections:

- Understanding File Formats
- Managing Log Files: the Log File Parameter
- Handling Message Channels: the Output-Channel Parameter
- Handling Message Channels: Callable Library Routines
- Example: Using the Message Handler

Understanding File Formats

The *ILOG CPLEX Reference Manual* documents the file formats that ILOG CPLEX supports. Here is a brief description of these file formats:

- BAS files are text files governed by MPS conventions (that is, they are not binary) for saving a simplex basis.
- BIN files are binary files. ILOG CPLEX uses this format when it writes solution files containing the binary representation of real numbers.
- DPE is the format ILOG CPLEX uses to write a problem in a binary SAV file after the objective of a problem has been perturbed for use with the dual simplex optimizer.
- DUA format, governed by MPS conventions, writes the dual formulation of a problem currently in memory so that the MPS file can later be read back in and the dual formulation can then be optimized explicitly. This file format is largely obsolete now since you can set the `PreDual` parameter to tell ILOG CPLEX to solve the dual formulation of an LP automatically. (You no longer have to tell ILOG CPLEX to write the dual formulation to a DUA file and then tell ILOG CPLEX to read the file back in and solve it.)
- EMB is the format ILOG CPLEX uses to save an embedded network it extracts from a problem. EMB files are written in MPS format.
- IIS is the format ILOG CPLEX uses to represent irreducible inconsistent sets of constraints. *Finding a Set of Irreducibly Inconsistent Constraints* on page 143 and *Example: Writing an IIS-Type File* on page 145 explain more about these kinds of files.
- LP (Linear Programming) is an ILOG CPLEX-specific file formatted for entering problems in an algebraic, row-oriented form. In other words, LP format allows you to enter problems in terms of their constraints. When you enter problems interactively in

the Interactive Optimizer, you are implicitly using LP format. ILOG CPLEX will also read in files in LP format. *Working with LP Files* on page 292 explains more fully how to use LP files with ILOG CPLEX.

- MIN format for representing minimum-cost network-flow problems was introduced by DIMACS in 1991. More information about DIMACS network file formats is available via anonymous ftp from `ftp://dimacs.rutgers.edu/pub/netflow/general-info/specs.tex`.
- MPS (Mathematical Programming System) is an industry-standard, ASCII-text file format for mathematical programming problems. Besides the industry conventions, ILOG CPLEX also supports extensions to this format for ILOG CPLEX-specific cases, such as names of more than eight characters, blank space as delimiters between columns, etc. *Working with MPS Files* on page 293 in this manual explains more fully how to use MPS files with ILOG CPLEX.
- MST is used to enter and save a starting solution for a MIP.
- NET is an ILOG CPLEX-specific ASCII format for network-flow problems. It is flexible and supports named nodes and arcs.
- ORD is a format used to enter and to save priority orders for branching. It may contain branching instructions for individual variables.
- PPE is the format ILOG CPLEX uses to write a problem in a binary SAV file after the bounds of a problem have been perturbed for use with the primal simplex optimizer.
- PRE is the format ILOG CPLEX uses to write a presolved, reduced problem formulation to a binary SAV file. Since a presolved problem has been reduced, it will not correspond to the original problem.
- QP is a format containing the values of nonzero coefficients in the $\boldsymbol{Q}$ matrix of a quadratic programming problem. You must enter the *linear* part of that quadratic programming problem *first* (by whichever means you choose).
- REW is used to write a problem in MPS format with disguised row and column names. This format may be useful, for example, for problems that you consider to be highly proprietary.
- SAV is an ILOG CPLEX-specific *binary* format for reading and writing problems and their associated basis information. ILOG CPLEX includes the basis in a SAV file only if the problem currently in memory has been optimized and a basis exists. This format offers the advantage of being numerically accurate (to the same degree as your platform) in contrast to text file formats that may lose numerical accuracy. It also has the additional benefit of being efficient with respect to read and write time. However, since a SAV file is binary, you cannot read nor edit it with a text editor.
- SOS is a format that declares special ordered sets, the set branching order, and weights for each set member.

- TRE is a format that saves information about progress through the branch & cut tree. It is a binary format.
- TXT files are ASCII-text files. ILOG CPLEX uses this format when it writes solution files in text.
- VEC is a format that saves the solution to a pure barrier optimization prior to crossover (that is, a nonbasis solution) that can later be read back in and used to initiate crossover. *Using VEC File Format* on page 160 explains how to use this file format.

Working with LP Files

LP files are row-oriented so you can look at a problem as you enter it in a naturally and intuitively algebraic way. However, ILOG CPLEX represents a problem internally in a column-ordered format. This difference between the way ILOG CPLEX accepts a problem in LP format and the way it stores the problem internally may have an impact on *memory use* and on the *order* in which *variables* are displayed on screen or in files.

Variable Order and LP Files

As ILOG CPLEX reads an LP format file by rows, it adds columns as it encounters them in a row. This convention will have an impact on the order in which variables are named and displayed. For example, consider this problem:

Maximize $2x_2 + 3x_3$

subject to

$-x_1 + x_2 + x_3 \leq 20$

$x_1 - 3x_2 + x_3 \leq 30$

with these bounds

$0 \leq x_1 \leq 40$

$0 \leq x_2 \leq +\infty$

$0 \leq x_3 \leq +\infty$

Since ILOG CPLEX reads the objective function as the first row, the two columns appearing there will become the first two variables. When the problem is displayed or rewritten into another LP file, the variables there will appear in a different order within each row. In this example, if you execute the command `display problem all`, you will see this:

```
Maximize
 obj: 2 x2 + 3 x3
Subject To
 c1: x2 + x3 - x1 <= 20
 c2: - 3 x2 + x3 + x1 <= 30
Bounds
 0 <= x1 <= 40
 All other variables are >= 0.
```

That is, x1 appears at the end of each constraint in which it has a nonzero coefficient. Also, while re-ordering like this does not affect the optimal objective function value of the problem, if there exist alternate optimal solutions at this value, then the different order of the variables could result in a change in the solution path of the algorithm, and there may be noticeable variation in the solution values of the individual variables.

Working with MPS Files

The ILOG CPLEX MPS file reader is highly compatible with existing modeling systems. There is generally no need to modify existing problem files to use them with ILOG CPLEX. However, there are ILOG CPLEX-specific conventions that may be useful for you to know. This section explains those conventions, and the *ILOG CPLEX Reference Manual* documents MPS format more fully.

Free Rows in MPS Files

In an MPS file, ILOG CPLEX selects the first free row or N-type row as the objective function, and it discards all subsequent free rows unless it is instructed otherwise by an OBJNAME section in the file. To retain free rows in an MPS file, reformulate them as equality rows with an additional free variable. For example, replace the free row `x + y` by the equality row `x + y - s = 0` where `s` is free. Generally, the ILOG CPLEX presolver will remove rows like that before optimization so they will have no impact on performance.

Ranged Rows in MPS Files

To handle ranged rows, ILOG CPLEX introduces a temporary range variable, creates appropriate bounds for this variable, and changes the sense of the row to an equality (that is, MPS type EQ). The added range variables will have the same name as the ranged row with the characters Rg prefixed. When ILOG CPLEX generates solution reports, it removes these temporary range variables from the constraint matrix.

Extra Rim Vectors in MPS Files

The MPS format allows multiple right-hand sides (RHSs), multiple bounds, and multiple range vectors. It also allows extra free rows. Together, these features are known as *extra rim vectors*. By default, the ILOG CPLEX MPS reader selects the first RHS, bound, and range definitions that it finds. The first free row (that is, N-type row) becomes the objective function, and the remaining free rows are discarded. The extra rim data are also discarded.

Naming Conventions in MPS Files

ILOG CPLEX accepts any noncontrol-character within a name. However, ILOG CPLEX recognizes blanks (that is, spaces) as delimiters, so you must avoid them in names. You should also avoid $ (dollar sign) and * (asterisk) as characters in names because they normally indicate a comment within a data record.

Error Checking in MPS Files

Fairly common problems in MPS files include split vectors, unnamed columns, and duplicated names. ILOG CPLEX checks for these conditions and reports them. If repeated

rows or columns occur in an MPS file, ILOG CPLEX reports an error and stops reading the file. You can then edit the MPS file to correct the source of the problem.

Saving Modified MPS Files

You may often want to save a modified MPS file for later use. To that end, ILOG CPLEX will write out a problem exactly as it appears in memory. All your revisions of that problem will appear in the new file. One potential area for confusion occurs when a maximization problem is saved. Since MPS conventionally represents all problems as minimizations, ILOG CPLEX reverses the sign of the objective-function coefficients when it writes a maximization problem to an MPS file. When you read and optimize this new problem, the values of the variables will be valid for the original model. However, since the problem has been converted from a maximization to the equivalent minimization, the objective, dual, and reduced-cost values will have reversed signs.

Converting File Formats

MPS, Mathematical Programming System, an industry-standard format based on ASCII-text has historically been restricted to a fixed format in which data fields were limited to eight characters and specific fields had to appear in specific columns on specific lines.
ILOG CPLEX supports extensions to MPS that allow more descriptive names (that is, more than eight characters), greater accuracy for numerical data, and greater flexibility in data positions.

Most MPS files in fixed format conform to the ILOG CPLEX extensions and thus can be read by the ILOG CPLEX MPS reader without error. However, the ILOG CPLEX MPS reader will *not* accept the following conventions:

- blank space within a name;
- blank lines;
- missing fields (such as bound names and right-hand side names);
- extraneous, uncommented characters;
- blanks in lieu of repeated name fields, such as bound vector names and right-hand side names.

You can convert fixed-format MPS files that contain those conventions into acceptable ILOG CPLEX-extended MPS files. To do so, use the `convert` utility supplied in the standard distribution of ILOG CPLEX. The `convert` utility removes unreadable features from fixed-format MPS, BAS, SOS, and ORD files. It runs from the operating system prompt of your platform. Here is the syntax of the convert utility:

```
convert -option inputfilename outputfilename
```

Your command must include an input-file name and an output-file name; they must be different from each other. The options, summarized in Table 9.1, indicate the file type. You

may specify only one option. If you do not specify an option, ILOG CPLEX attempts to deduce the file type from the extension in the file name.

Table 9.1 *Options for the* `convert` *Utility and Corresponding File Extensions*

Option	File type	File extension
`-m`	MPS (Mathematical Programming System)	`.mps`
`-s`	SOS (Special Ordered Set)	`.sos`
`-b`	BAS (basis file according to MPS conventions)	`.bas`
`-o`	ORD (priority orders)	`.ord`

Managing Log Files: the Log File Parameter

As ILOG CPLEX is working, it can record messages to a log file. By default, the Interactive Optimizer creates the log file in the directory where it is running, and it names the file `cplex.log`. If such a file already exists, ILOG CPLEX adds a line indicating the current time and date and then appends new information to the end of the existing file. That is, it does not overwrite the file, and it distinguishes different sessions within the log file. By default, there is no log file for Component Library applications.

You can locate the log file where you like, and you can rename it. Some users, for example, like to create a specifically named log file for each session. Also you can close the log file in case you do not want ILOG CPLEX to record messages to its default log file.

The following sections show you the commands for creating, renaming, relocating, and closing a log file.

Creating, Renaming, Relocating Log Files

- In the Interactive Optimizer, use the command `set logfile filename`, substituting the name you prefer for the log file. In other words, use this command to rename or relocate the default log file.
- From the Callable Library, first use the routine `CPXfopen()` to open the target file; then use the routine `CPXsetlogfile()`. The *ILOG CPLEX Reference Manual* documents both routines.
- From Concert, use the `setOut()` method to send logging output to the specified output stream.

Closing Log Files

- If you do not want ILOG CPLEX to record messages in a log file, then you can close the log file from the Interactive Optimizer with the command `set logfile *`.

- By default, routines from the Callable Library do not write to a log file. However, if you want to close a log file that you created by a call to `CPXsetlogfile()`, call `CPXsetlogfile()` again, and this time, pass a `NULL` pointer as its second argument.
- From Concert, use the `setOut()` method with `env.getNullStream()` as argument, where `env` is an `IloEnv` object, to stop sending logging output to an output stream.

Handling Message Channels: the Output-Channel Parameter

Besides the log-file parameter, Interactive Optimizer and the Callable Library offer you output-channel parameters to give you finer control over when and where messages appear in the Interactive Optimizer. Output-channel parameters indicate whether output should or should not appear on screen. They also allow you to designate log files for message channels. The output-channel parameters do not affect the log-file parameter, so it is customary to use the command `set logfile` before the command `set output channel value1 value2`.

In the output-channel command, you can specify a `channel` to be one of `dialog`, `errors`, `logonly`, `results`, or `warnings`. Table 9.2 summarizes the information carried over each channel.

Table 9.2 *Options for the Output-Channel Command*

Channel	Information
`dialog`	messages related to interactive use; e.g., prompts, help messages, greetings
`errors`	messages to inform user that operation could not be performed and why
`logonly`	message to record only in file (not on screen) e.g., multiline messages
`results`	information explicitly requested by user; state, change, progress information
`warnings`	messages to inform user request was performed but unexpected condition may result

The option `value2` lets you specify a file name to redirect output from a channel.

Also in that command, `value1` allows you to turn on or off output to the screen. When `value1` is `y`, output is directed to the screen; when its value is `n`, output is not directed to the screen. Table 9.3 summarizes which channels direct output to the screen by default. If a

channel directs output to the screen by default, you can leave `value1` blank to get the same effect as `set output channel y`.

Table 9.3 *Channels Directing Output to Screen or to a File*

Channel	Default value1	Meaning
`dialog`	`y`	blank directs output to screen but not to a file
`errors`	`y`	blank directs output to screen and to a file
`logonly`	`n`	blank directs output only to a file, not to screen
`results`	`y`	blank directs output to screen and to a file
`warnings`	`y`	blank directs output to screen and to a file

Handling Message Channels: Callable Library Routines

Interactive Optimizer and the Callable Library define several message channels for flexible control over message output:

- `cpxresults` for messages containing status and progress information;
- `cpxerror` for messages issued when a task cannot be completed;
- `cpxwarning` for messages issued when a nonfatal difficulty is encountered; or when an action taken may have side-effects; or when an assumption made may have side-effects;
- `cpxlog` for messages containing information that would not conventionally be displayed on screen but could be useful in a log file.

Output messages flow through message channels to destinations. Message channels are associated with destinations through their destination list. Messages from routines of the ILOG CPLEX Callable Library are assigned internally to one of those predefined channels. Those default channels are C pointers to ILOG CPLEX objects; they are initialized by `CPXopenCPLEX()`; they are *not* global variables. Your application accesses these objects by calling the routine `CPXgetchannels()`. You can use these predefined message channels for your own application messages. You can also define new channels.

An application using routines from the ILOG CPLEX Callable Library produces no output messages unless the application specifies message handling instructions through one or more calls to the message handling routines of the Callable Library. In other words, the destination list of each channel is initially empty.

Messages from multiple channels may be sent to one destination. All predefined ILOG CPLEX channels can be directed to a single file by a call to `CPXsetlogfile()`. Similarly, all predefined ILOG CPLEX channels except `cpxlog` can be directed to the screen using the `CPX_PARAM_SCRIND` parameter. For a finer level of control, or to define

destinations for application-specific messages, use the following message handling routines, all documented in the *ILOG CPLEX Reference Manual*:

- `CPXmsg()` writes a message to a predefined channel;
- `CPXflushchannel()` flushes a channel to its associated destination;
- `CPXdisconnectchannel()` flushes a channel and clears its destination list;
- `CPXdelchannel()` flushes a channel, clears its destination list, frees memory for that channel;
- `CPXaddchannel()` adds a channel;
- `CPXaddfpdest()` adds a destination file to the list of destinations associated with a channel;
- `CPXdelfpdest()` deletes a destination from the destination list of a channel;
- `CPXaddfuncdest()` adds a destination function to a channel;
- `CPXdelfuncdest()` deletes a destination function to a channel;

Once channel destinations are established, messages can be sent to multiple destinations by a single call to a message-handling routine.

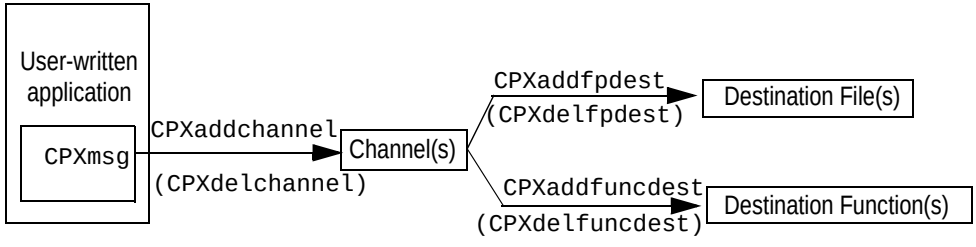

Figure 9.1 *ILOG CPLEX Message Handling Routines*

Example: Using the Message Handler

This example shows you how to use the ILOG CPLEX message handler from the Callable Library. It captures all messages generated by ILOG CPLEX and displays them on screen along with a label indicating which channel sent the message. It also creates a user channel to receive output generated by the program itself. The user channel accepts user-generated messages, displays them on screen with a label, and records them in a file without the label.

This example derives from `lpex1.c`, a program described in the ILOG CPLEX *Getting Started* manual. There are a few differences between the two examples:

- In this example, the function `ourmsgfunc()`—rather than the C functions `printf()` or `fprintf(stderr, . . .)`—manages all output. The program itself or `CPXmsg()` from the ILOG CPLEX Callable Library calls `ourmsgfunc()`. In fact, `CPXmsg()` is a replacement for `printf()`, allowing a message to appear in *more than one place*, for example, both on screen and in a file.

  Only *after* you initialize the ILOG CPLEX environment by calling `CPXopenCPLEX()` can you call `CPXmsg()`. And only *after* you call `CPXgetchannels()` can you use the default ILOG CPLEX channels. Therefore, calls to `ourmsgfunc()` print directly any messages that occur *before* the program gets the address of `cpxerror` (a channel). After a call to `CPXgetchannels()` gets the address of `cpxerror`, and after a call to `CPXaddfuncdest()` associates the message function `ourmsgfunc()` with `cpxerror`, then error messages are generated by calls to `CPXmsg()`.

  After the `TERMINATE:` label, any error must be generated with care in case the error message function has not been set up properly. Thus, `ourmsgfunc()` is also called directly to generate any error messages there.

- A call to the ILOG CPLEX Callable Library routine `CPXaddchannel()` initializes the channel `ourchannel`. The Callable Library routine `fopen()` opens the file `lpex5.out` to accept solution information. A call the ILOG CPLEX Callable Library routine `CPXaddfpdest()` associates that file with that channel. Solution information is also displayed on screen since `ourmsgfunc()` is associated with that new channel, too. Thus in the loops near the end of `main()`, when the solution is printed, only one call to `CPXmsg()` suffices to put the output both on screen and into the file. A call to `CPXdelchannel()` deletes `ourchannel`.

- Although `CPXcloseCPLEX()` will automatically delete file- and function-destinations for channels, we recommend that you call `CPXdelfpdest()` and `CPXdelfuncdest()` as the end of your programs.

Complete Program: lpex5.c

The complete program, `lpex5.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex5.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex5.c - Illustrating the CPLEX message handler
   This is a modification of lpex1.c .
   We'll label each channels output with its name, and
   also put our own output into a file.  Note that the labels
   may cause the output to go past 80 characters per line. */
```

```
/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static void CPXPUBLIC
   ourmsgfunc     (void *handle, const char *message);



/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6


int
main (void)
{
   char     probname[16];  /* Problem name is max 16 characters */

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;
   char          errmsg[1024];

   CPXCHANNELptr  cpxerror   = NULL;
   CPXCHANNELptr  cpxwarning = NULL;
   CPXCHANNELptr  cpxresults = NULL;
   CPXCHANNELptr  ourchannel = NULL;

   char *errorlabel = "cpxerror";
```

```
char *warnlabel  = "cpxwarning";
char *reslabel   = "cpxresults";
char *ourlabel   = "Our Channel";

CPXFILEptr fpout  = NULL;


/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

/* Since the message handler is yet to be set up, we'll call our
   messaging function directly to print out any errors  */

if ( env == NULL ) {
   /* The message argument for ourmsgfunc must not be a constant,
      so copy the mesage to a buffer. */
   strcpy (errmsg, "Could not open CPLEX environment.\n");
   ourmsgfunc ("Our Message", errmsg);
   goto TERMINATE;
}

/* Now get the standard channels.  If an error, just call our
   message function directly. */

status = CPXgetchannels (env, &cpxresults, &cpxwarning, &cpxerror, NULL);
if ( status ) {
   strcpy (errmsg, "Could not get standard channels.\n");
   ourmsgfunc ("Our Message", errmsg);
   CPXgeterrorstring (env, status, errmsg);
   ourmsgfunc ("Our Message", errmsg);
   goto TERMINATE;
}

/* Now set up the error channel first.  The label will be "cpxerror" */

status = CPXaddfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
if ( status ) {
   strcpy (errmsg, "Could not set up error message handler.\n");
   ourmsgfunc ("Our Message", errmsg);
   CPXgeterrorstring (env, status, errmsg);
   ourmsgfunc ("Our Message", errmsg);
}

/* Now that we have the error message handler set up, all CPLEX
   generated errors will go through ourmsgfunc.  So we don't have
   to use CPXgeterrorstring to determine the text of the message.
   We can also use CPXmsg to do any other printing.  */

status = CPXaddfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
if ( status ) {
```

```
   CPXmsg (cpxerror, "Failed to set up handler for cpxwarning.\n");
   goto TERMINATE;
}

status = CPXaddfuncdest (env, cpxresults, reslabel, ourmsgfunc);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up handler for cpxresults.\n");
   goto TERMINATE;
}

/* Now turn on the iteration display. */

status = CPXsetintparam (env, CPX_PARAM_SIMDISPLAY, 2);
if ( status ) {
   CPXmsg (cpxerror, "Failed to turn on simplex display level.\n");
   goto TERMINATE;
}

/* Create the problem. */

strcpy (probname, "example");
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  */

if ( lp == NULL ) {
   CPXmsg (cpxerror, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data. */

status = populatebycolumn (env, lp);

if ( status ) {
   CPXmsg (cpxerror, "Failed to populate problem data.\n");
   goto TERMINATE;
}


/* Optimize the problem and obtain solution. */

status = CPXlpopt (env, lp);
if ( status ) {
   CPXmsg (cpxerror, "Failed to optimize LP.\n");
   goto TERMINATE;
}

status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
if ( status ) {
   CPXmsg (cpxerror, "Failed to obtain solution.\n");
   goto TERMINATE;
}
```

```
/* Write the output to the screen.  We will also write it to a
   file as well by setting up a file destination and a function
   destination. */

ourchannel = CPXaddchannel (env);
if ( ourchannel == NULL ) {
   CPXmsg (cpxerror, "Failed to set up our private channel.\n");
   goto TERMINATE;
}

fpout = CPXfopen ("lpex5.msg", "w");
if ( fpout == NULL ) {
   CPXmsg (cpxerror, "Failed to open lpex5.msg file for output.\n");
   goto TERMINATE;
}
status = CPXaddfpdest (env, ourchannel, fpout);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up output file destination.\n");
   goto TERMINATE;
}

status = CPXaddfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
if ( status ) {
   CPXmsg (cpxerror, "Failed to set up our output function.\n");
   goto TERMINATE;
}

/* Now any message to channel ourchannel will go into the file
   and into the file opened above. */

CPXmsg (ourchannel, "\nSolution status = %d\n", solstat);
CPXmsg (ourchannel, "Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using sizes from when the problem
   was built.  cur_numrows and cur_numcols store the current number
   of rows and columns, respectively.  */

cur_numrows = CPXgetnumrows (env, lp);
cur_numcols = CPXgetnumcols (env, lp);
for (i = 0; i < cur_numrows; i++) {
   CPXmsg (ourchannel, "Row %d:  Slack = %10f  Pi = %10f\n",
           i, slack[i], pi[i]);
}

for (j = 0; j < cur_numcols; j++) {
   CPXmsg (ourchannel, "Column %d:  Value = %10f  Reduced cost = %10f\n",
           j, x[j], dj[j]);
}

/* Finally, write a copy of the problem to a file. */

status = CPXwriteprob (env, lp, "lpex5.lp", NULL);
if ( status ) {
   CPXmsg (cpxerror, "Failed to write LP to disk.\n");
   goto TERMINATE;
```

```
   }


TERMINATE:

   /* First check if ourchannel is open */

   if ( ourchannel != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, ourchannel, ourlabel, ourmsgfunc);
      if ( chanstat ) {
         strcpy (errmsg, "CPXdelfuncdest failed.\n");
         ourmsgfunc ("Our Message", errmsg);
         if (!status)  status = chanstat;
      }
      if ( fpout != NULL ) {
         chanstat = CPXdelfpdest (env, ourchannel, fpout);
         if ( chanstat ) {
            strcpy (errmsg, "CPXdelfpdest failed.\n");
            ourmsgfunc ("Our Message", errmsg);
            if (!status)  status = chanstat;
         }
         CPXfclose (fpout);
      }

      CPXdelchannel (env, &ourchannel);
   }

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         strcpy (errmsg, "CPXfreeprob failed.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   /* Now delete our function destinations from the 3 CPLEX channels. */
   if ( cpxresults != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxresults, reslabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxresults function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   if ( cpxwarning != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxwarning, warnlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxwarning function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }
```

```
   if ( cpxerror != NULL ) {
      int  chanstat;
      chanstat = CPXdelfuncdest (env, cpxerror, errorlabel, ourmsgfunc);
      if ( chanstat && !status ) {
         status = chanstat;
         strcpy (errmsg, "Failed to delete cpxerror function.\n");
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         strcpy (errmsg, "Could not close CPLEX environment.\n");
         ourmsgfunc ("Our Message", errmsg);
         CPXgeterrorstring (env, status, errmsg);
         ourmsgfunc ("Our Message", errmsg);
      }
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
```

```
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;

   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* For our message functions, we will interpret the handle as a pointer
 * to a string, which will be the label for the channel.  We'll put
 * angle brackets <> around the message so its clear what the function is
 * sending to us.  We'll place the newlines that appear at the end of
 * a message after the > bracket.  The 'message' argument must not be
 * a constant, since it is changed by this function.
 */

static void CPXPUBLIC
```

```
ourmsgfunc (void *handle, const char *msg)
{
   char  *label;
   int   lenstr;
   int   flag = 0;
   char  *message = (char *)msg;

   lenstr = strlen(message);
   if ( message[lenstr-1] == '\n' ) {
      message[lenstr-1] = '\0';
      flag = 1;
   }

   label = (char *) handle;
   printf ("%-15s: <%s>", label, message);
   if (flag) putchar('\n');

   /* If we clobbered the '\n', we need to put it back */

   if ( flag )  message[lenstr-1] = '\n';

} /* END ourmsgfunc */
```

Using Query Routines in the Callable Library

This section tells you how to use the Callable Library query routines. In Concert Technology (for C++ and Java users), query methods such as `getName()` use objects that can be referenced directly. Thus this section concentrates on the Callable Library routines only. However, equivalent Concert Technology examples are also provided in the distribution.

Using Surplus Arguments for Array Allocations

Most of the ILOG CPLEX query routines in the Callable Library require your application to allocate memory for one or more arrays that will contain the results of the query. In many cases, your application—the calling program—does not know the size of these arrays in advance. For example, in a call to `CPXgetcols()` requesting the matrix data for a range of columns, your application needs to pass the arrays `cmatind` and `cmatval` for ILOG CPLEX to populate with matrix coefficients and row indices. However, unless your application has carefully kept track of the number of nonzeros in each column throughout the problem specification and, if applicable, throughout its modification, the actual length of these arrays remains unknown.

Fortunately, the ILOG CPLEX query routines in the Callable Library contain a `surplus_p` argument that, when used in conjunction with the array length arguments, enables you first to call the query routine to determine the length of the required array. Then, when the length is known, your application can properly allocate these arrays. Afterwards, your application makes a second call to the query routine with the correct array lengths to obtain the requested data.

For example, consider a program that needs to call `CPXgetcols()` to access a range of columns. Here is the list of arguments for `CPXgetcols()`.

```
CPXgetcols (CPXENVptr env,
            CPXLPptr lp,
            int *nzcnt_p,
            int *cmatbeg,
            int *cmatind,
            double *cmatval,
            int cmatspace,
            int *surplus_p,
            int begin,
            int end);
```

The arrays `cmatind` and `cmatval` require one element for each nonzero matrix coefficient in the requested range of columns. The required length of these arrays, specified in `cmatspace`, remains unknown at the time of the query. Your application—the calling program—can determine the length of these arrays by first calling `CPXgetcols()` with a value of `0` for `cmatspace`. This call will return an error status of `CPXERR_NEGATIVE_SURPLUS` indicating a shortfall of the array length specified in `cmatspace` (in this case, `0`); it will also return the actual number of matrix nonzeros in the

requested range of columns. CPXgetcols() deposits this shortfall as a negative number in the integer pointed to by surplus_p. Your application can then negate this shortfall and allocate the arrays cmatind and cmatval sufficiently long to contain all the requested matrix elements.

The following sample of code shows you what we mean. The first call to CPXgetcols() passes a value of 0 for cmatspace in order to obtain the shortfall in cmatsz. The sample then uses the shortfall to allocate the arrays cmatind and cmatval properly; then it calls CPXgetcols() again to obtain the actual matrix coefficients and row indices.

```
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, NULL, NULL,
                     0, &cmatsz, 0, numcols - 1);
if ( status != CPXERR_NEGATIVE_SURPLUS ) {
   if ( status != 0 ) {
      CPXmsg (cpxerror,
              "CPXgetcols for surplus failed, status = %d\n", status);
      goto TERMINATE;
   }
   CPXmsg (cpxwarning,
           "All columns in range [%d, %d] are empty.\n",
           0, (numcols - 1));
}
cmatsz   = -cmatsz;
cmatind  = (int *) malloc ((unsigned) (1 + cmatsz)*sizeof(int));
cmatval  = (double *) malloc ((unsigned) (1 + cmatsz)*sizeof(double));
if ( cmatind == NULL || cmatval == NULL ) {
   CPXmsg (cpxerror, "CPXgetcol mallocs failed\n");
   status = 1;
   goto TERMINATE;
}
status = CPXgetcols (env, lp, &nzcnt, cmatbeg, cmatind, cmatval,
                     cmatsz, &surplus, 0, numcols - 1);
if ( status ) {
   CPXmsg (cpxerror, "CPXgetcols failed, status = %d\n", status);
   goto TERMINATE;
}
```

That sample code (or your application) does not need to determine the length of the array cmatbeg. The array cmatbeg has one element for each column in the requested range. Since this length is known ahead of time, your application does not need to call a query routine to calculate it. More generally, query routines use surplus arguments in the way we just described only for the length of any array required to store problem data of unknown length. Problem data in this category includes nonzero matrix entries, row and column names, other problem data names, special ordered sets (SOS), priority orders, and MIP start information.

Example: Using Query Routines

This example uses the ILOG CPLEX Callable Library *query routine* CPXgetcolname() to get the column names from a problem object. To do so, it applies the programming pattern we just described in *Using Surplus Arguments for Array Allocations* on page 308. It derives

from the example `lpex2.c`, explained in the manual ILOG CPLEX *Getting Started* manual. This query-routine example differs from that simpler example in several ways:

- The example calls `CPXgetcolname()` twice after optimization: the first call determines how much space to allocate to hold the names; the second call gets the names and stores them in the arrays `cur_colname` and `cur_colnamestore`.
- When the example prints its answer, it uses the names as stored in `cur_colname`. If no names exist there, the example creates generic names.

This example assumes that the current problem has been read from a file by `CPXreadcopyprob()`. You can adapt the example to use other ILOG CPLEX query routines to get information about any problem read from a file.

Complete Program: lpex7.c

The complete program, `lpex7.c`, appears here and online in the standard distribution.

```
/*--------------------------------------------------------------------------*/
/*  File: examples/src/lpex7.c                                              */
/*  Version 8.1                                                             */
/*--------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                        */
/*  All Rights Reserved.                                                    */
/*  Permission is expressly granted to use this example in the              */
/*  course of developing applications that use ILOG products.               */
/*--------------------------------------------------------------------------*/

/* lpex7.c - Reading in and optimizing a problem.  Printing
             names with the answer.  This is a modification of
             lpex2.c  */

/* To run this example, command line arguments are required.
   i.e.,   lpex7   filename   method
   where
       filename is the name of the file, with .mps, .lp, or .sav extension
       method   is the optimization method
                   o          default
                   p          primal simplex
                   d          dual   simplex
                   n          network with dual simplex cleanup
                   h          barrier with crossover
                   b          barrier without crossover
                   s          sifting
                   c          concurrent
   Example:
       lpex7  example.mps  o
 */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */
```

```
#include <ilcplex/cplex.h>

/* Bring in the declarations for the string and character functions
   and malloc */

#include <ctype.h>
#include <stdlib.h>
#include <string.h>

/* Include declarations for functions in this program */

static void
   free_and_null (char **ptr),
   usage         (char *progname);


int
main (int argc, char *argv[])
{
   /* Declare and allocate space for the variables and arrays where we will
      store the optimization results including the status, objective value,
      maximum bound violation, variable values, and basis. */

   int      solnstat, solnmethod, solntype;
   double   objval, maxviol;
   double   *x     = NULL;
   int      *cstat = NULL;
   int      *rstat = NULL;

   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status = 0;
   int           j;
   int           cur_numrows, cur_numcols;
   char          **cur_colname = NULL;
   char          *cur_colnamestore = NULL;
   int           cur_colnamespace;
   int           surplus;
   int           method;

   char          *basismsg;

   /* Check the command line arguments */

   if (( argc != 3 )                                 ||
       ( strchr ("podhbnsc", argv[2][0]) == NULL )  ) {
      usage (argv[0]);
      goto TERMINATE;
   }

   /* Initialize the CPLEX environment */
```

```
env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  A call to CPXgeterrorstring will produce the text of
   the error message.  Note that CPXopenCPLEX produces no output,
   so the only way to see the cause of the error is to use
   CPXgeterrorstring.  For other CPLEX routines, the errors will
   be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON.  */

if ( env == NULL ) {
   char  errmsg[1024];
   fprintf (stderr, "Could not open CPLEX environment.\n");
   CPXgeterrorstring (env, status, errmsg);
   fprintf (stderr, "%s", errmsg);
   goto TERMINATE;
}

/* Turn on output to the screen */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_ON);
if ( status ) {
   fprintf (stderr,
            "Failure to turn on screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem, using the filename as the problem name */

lp = CPXcreateprob (env, &status, argv[1]);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, the setting of
   the parameter CPX_PARAM_SCRIND causes the error message to
   appear on stdout.  Note that most CPLEX routines return
   an error code to indicate the reason for failure.   */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now read the file, and copy the data into the created lp */

status = CPXreadcopyprob (env, lp, argv[1], NULL);
if ( status ) {
   fprintf (stderr, "Failed to read and copy the problem data.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */
```

```
   switch (argv[2][0]) {
      case 'o':
         method = CPX_ALG_AUTOMATIC;
         break;
      case 'p':
         method = CPX_ALG_PRIMAL;
         break;
      case 'd':
         method = CPX_ALG_DUAL;
         break;
      case 'n':
         method = CPX_ALG_NET;
         break;
      case 'h':
         method = CPX_ALG_BARRIER;
         break;
      case 'b':
         method = CPX_ALG_BARRIER;
         status = CPXsetintparam (env, CPX_PARAM_BARCROSSALG, CPX_ALG_NONE);
         if ( status ) {
            fprintf (stderr,
                     "Failed to set the crossover method, error %d.\n",
status);
            goto TERMINATE;
         }
         break;
      case 's':
         method = CPX_ALG_SIFTING;
         break;
      case 'c':
         method = CPX_ALG_CONCURRENT;
         break;
      default:
         method = CPX_ALG_NONE;
         break;
   }

   status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, method);
   if ( status ) {
      fprintf (stderr,
               "Failed to set the optimization method, error %d.\n", status);
      goto TERMINATE;
   }


   status = CPXlpopt (env, lp);
   if ( status ) {
      fprintf (stderr, "Failed to optimize LP.\n");
      goto TERMINATE;
   }

   solnstat = CPXgetstat (env, lp);
```

```
if      ( solnstat == CPX_STAT_UNBOUNDED ) {
   printf ("Model is unbounded\n");
   goto TERMINATE;
}
else if ( solnstat == CPX_STAT_INFEASIBLE ) {
   printf ("Model is infeasible\n");
   goto TERMINATE;
}
else if ( solnstat == CPX_STAT_INForUNBD ) {
   printf ("Model is infeasible or unbounded\n");
   goto TERMINATE;
}

status = CPXsolninfo (env, lp, &solnmethod, &solntype, NULL, NULL);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution info.\n");
   goto TERMINATE;
}
printf ("Solution status %d, solution method %d\n", solnstat, solnmethod);

if ( solntype == CPX_NO_SOLN ) {
   fprintf (stderr, "Solution not available.\n");
   goto TERMINATE;
}

status = CPXgetobjval (env, lp, &objval);
if ( status ) {
   fprintf (stderr, "Failed to obtain objective value.\n");
   goto TERMINATE;
}
printf ("Objective value %.10g.\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is.  cur_numrows and cur_numcols store the
   current number of rows and columns, respectively.  */

cur_numcols = CPXgetnumcols (env, lp);
cur_numrows = CPXgetnumrows (env, lp);

/* Retrieve basis, if one is available */

if ( solntype == CPX_BASIC_SOLN ) {
   cstat = (int *) malloc (cur_numcols*sizeof(int));
   rstat = (int *) malloc (cur_numrows*sizeof(int));
   if ( cstat == NULL || rstat == NULL ) {
      fprintf (stderr, "No memory for basis statuses.\n");
      goto TERMINATE;
   }

   status = CPXgetbase (env, lp, cstat, rstat);
   if ( status ) {
      fprintf (stderr, "Failed to get basis; error %d.\n", status);
```

```
      goto TERMINATE;
   }
}
else {
   printf ("No basis available\n");
}


/* Retrieve solution vector */

x = (double *) malloc (cur_numcols*sizeof(double));
if ( x == NULL ) {
   fprintf (stderr, "No memory for solution.\n");
   goto TERMINATE;
}

status = CPXgetx (env, lp, x, 0, cur_numcols-1);
if ( status ) {
   fprintf (stderr, "Failed to obtain primal solution.\n");
   goto TERMINATE;
}


/* Now get the column names for the problem.  First we determine how
   much space is used to hold the names, and then do the allocation.
   Then we call CPXgetcolname() to get the actual names. */

status = CPXgetcolname (env, lp, NULL, NULL, 0, &surplus, 0,
                        cur_numcols-1);

if (( status != CPXERR_NEGATIVE_SURPLUS ) &&
    ( status != 0 )                          )  {
   fprintf (stderr,
            "Could not determine amount of space for column names.\n");
   goto TERMINATE;
}

cur_colnamespace = - surplus;
if ( cur_colnamespace > 0 ) {
   cur_colname      = (char **) malloc (sizeof(char *)*cur_numcols);
   cur_colnamestore = (char *)  malloc (cur_colnamespace);
   if ( cur_colname      == NULL ||
        cur_colnamestore == NULL   ) {
      fprintf (stderr, "Failed to get memory for column names.\n");
      status = -1;
      goto TERMINATE;
   }
   status = CPXgetcolname (env, lp, cur_colname, cur_colnamestore,
                           cur_colnamespace, &surplus, 0, cur_numcols-1);
   if ( status ) {
      fprintf (stderr, "CPXgetcolname failed.\n");
      goto TERMINATE;
```

```
      }
   }
   else {
      printf ("No names associated with problem.  Using Fake names.\n");
   }

   /* Write out the solution */

   for (j = 0; j < cur_numcols; j++) {
      if ( cur_colnamespace > 0 ) {
         printf ("%-16s:  ", cur_colname[j]);
      }
      else {
         printf ("Fake%-6.6d      :  ", j);;
      }
      printf ("%17.10g", x[j]);
      if ( cstat != NULL ) {
         switch (cstat[j]) {
            case CPX_AT_LOWER:
               basismsg = "Nonbasic at lower bound";
               break;
            case CPX_BASIC:
               basismsg = "Basic";
               break;
            case CPX_AT_UPPER:
               basismsg = "Nonbasic at upper bound";
               break;
            case CPX_FREE_SUPER:
               basismsg = "Superbasic, or free variable at zero";
               break;
            default:
               basismsg = "Bad basis status";
               break;
         }
         printf ("  %s",basismsg);
      }
      printf ("\n");
   }

   /* Display the maximum bound violation. */

   status = CPXgetdblquality (env, lp, &maxviol, CPX_MAX_PRIMAL_INFEAS);
   if ( status ) {
      fprintf (stderr, "Failed to obtain bound violation.\n");
      goto TERMINATE;
   }
   printf ("Maximum bound violation = %17.10g\n", maxviol);


TERMINATE:

   /* Free up the basis and solution */
```

```
   free_and_null ((char **) &cstat);
   free_and_null ((char **) &rstat);
   free_and_null ((char **) &x);
   free_and_null ((char **) &cur_colname);
   free_and_null ((char **) &cur_colnamestore);

   /* Free up the problem, if necessary */

   if ( lp != NULL ) {
      status = CPXfreeprob (env, &lp);
      if ( status ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", status);
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      status = CPXcloseCPLEX (&env);

      /* Note that CPXcloseCPLEX produces no output,
         so the only way to see the cause of the error is to use
         CPXgeterrorstring.  For other CPLEX routines, the errors will
         be seen if the CPX_PARAM_SCRIND indicator is set to CPX_ON. */

      if ( status ) {
         char  errmsg[1024];
         fprintf (stderr, "Could not close CPLEX environment.\n");
         CPXgeterrorstring (env, status, errmsg);
         fprintf (stderr, "%s", errmsg);
      }
   }

   return (status);

}  /* END main */



/* This simple routine frees up the pointer *ptr, and sets *ptr to NULL */

static void
free_and_null (char **ptr)
{
   if ( *ptr != NULL ) {
      free (*ptr);
      *ptr = NULL;
   }
} /* END free_and_null */
```

```
static void
usage (char *progname)
{
   fprintf (stderr,"Usage: %s filename algorithm\n", progname);
   fprintf (stderr,"   where filename is a file with extension \n");
   fprintf (stderr,"      MPS, SAV, or LP (lower case is allowed)\n");
   fprintf (stderr,"   and algorithm is one of the letters\n");
   fprintf (stderr,"      o          default\n");
   fprintf (stderr,"      p          primal simplex\n");
   fprintf (stderr,"      d          dual simplex\n");
   fprintf (stderr,"      n          network simplex\n");
   fprintf (stderr,"      b          barrier\n");
   fprintf (stderr,"      h          barrier with crossover\n");
   fprintf (stderr,"      s          sifting\n");
   fprintf (stderr,"      c          concurrent\n");
   fprintf (stderr," Exiting...\n");
} /* END usage */
```

Goals in IloCplex

Goals allow you to take control of the branch & cut search procedure used by `IloCplex` to solve MIP problems. To help you understand how to use goals with `IloCplex`, we review how this procedure works.

The `IloCplex` branch & cut search procedure manages a search tree consisting of nodes. Every node represents a subproblem to be solved, and the root node of the tree represents the entire problem. Nodes are called *active* if they have not yet been processed.

The tree is first initialized to contain the root node as the only active node. `IloCplex` processes active nodes from the tree until either no more active nodes are available or some limit has been reached. Once a node has been processed, it is no longer active.

When processing a node, `IloCplex` starts by solving the continuous relaxation of its subproblem—that is, the subproblem without integrality constraints. If the solution violates any cuts, `IloCplex` adds them to the node problem and re-solves. This is iterated until no more violated cuts are found by `IloCplex`. If at any point the relaxation becomes infeasible, the node is pruned, that is, it is removed from the tree.

To solve the node problem, `IloCplex` checks whether the solution satisfies the integrality constraints. If so, and if its objective value is better than that of the current incumbent, the solution of the node problem is used as the new incumbent. Otherwise, `IloCplex` splits the node problem into one or two smaller subproblems, typically by branching on a variable that violates its integrality condition. These subproblems are added to the tree as active nodes and the current node is deactivated.

The primary use of goals is to take control of these last steps, namely the feasibility test and the creation of subproblems. However, as we will discuss later, goals also allow you to add local and global cuts.

> ***Note:*** *The discussion of the details of using goals will be presented mainly in terms of the C++ library. The Java library follows the same design and is thus equivalent. In cases where a difference between the C++ and Java API needs to be observed, the point will be raised. Where the difference is only in syntax, the Java syntax will be mentioned in parentheses following the C++ syntax*

In C++, goals are implemented in objects of type `IloCplex::GoalI` (having handle class `IloCplex::Goal`). In Java, goals are implemented in objects of type `IloCplex.Goal` (and there are no handle classes). The method `IloCplex::GoalI::execute()` (`IloCplex.Goal.execute()`) is where all the control is implemented. This method is called by `IloCplex` after a node relaxation has been solved and all cuts have been added. Invoking method `execute()` of a goal is often referred to as *executing a goal*. When method `execute()` is executed, the member functions of class `IloCplex::GoalI`

(`IloCplex.Goal`) can be called to query information about the current node problem and the solution of its relaxation.

Typically, the implementation of method `execute()` will perform the following steps:

1. Check feasibility. An interesting possibility here is that the feasibility check may include more than verifying integrality of the solution. This allows you to enforce constraints that could not reasonably be expressed using linear constraints through cuts or branching. In other words, this allows you to use goals in a way that makes them part of the model to be solved. Such a use is common in Constraint Programming, but it is less frequently used in Mathematical Programming.
2. Optionally find local or global cuts to be added. Local cuts will be respected only for the subtree below the current node, whereas global cuts will be enforced for all nodes from then on.
3. Optionally construct a solution and pass it to `IloCplex`.
4. Instruct `IloCplex` how to proceed. Instructing `IloCplex` how to proceed is done through the return value of method `execute()`, which is another goal. `IloCplex` simply continues by executing this goal.

`IloCplex` provides a selection of special goals that can be used to specify how to proceed:

- Method `IloCplex::GoalI::OrGoal()` (`IloCplex.or()`) returns a goal that creates subnodes of the current node. This function takes at least 2 and up to 6 goals as parameters. For each of its parameters, the `Or` goal will create a subnode in such a way that when processing that subnode, the corresponding goal will be executed. Once the goal has been executed, the current node is immediately deactivated.
- Method `IloCplex::GoalI::AndGoal()` (`IloCplex.and()`) also takes goals as parameters. It returns a goal that will cause `IloCplex` to execute the goals passed as parameters in the order of the parameters.
- Method `IloCplex::GoalI::FailGoal()` (`IloCplex.failGoal()`) creates a goal that causes `IloCplex` to prune the current node. In other words, it discontinues the search at the node where the goal is executed. `IloCplex` will continue with another active node from the tree, if available.
- Class `IloCplex::Goal` has constructors that take an `IloRange` object or an `IloRangeArray` (`IloRange[]`)object as parameters. When one of these constructors is used, a *local cut goal* is created. Local cut goals add local cuts to the node where they are executed. To create local cut goals with the Java API, use method `IloCplex.constraintGoal()` or if more convenient, one of the methods `IloCplex.leGoal()`, `IloCplex.geGoal()` or `IloCplex.eqGoal()`.

- The 0 (`null`) goal, or empty goal, that is, an `IloCplex::Goal` handle object with 0 implementation pointer can also be returned by method `IloCplex::GoalI::execute()`. In most cases this will instruct `IloCplex` to take over control of the branch & cut search with its built-in strategies.

Since `IloCplex::GoalI::OrGoal()` (`IloCplex.or()`) and `IloCplex::GoalI::AndGoal()` (`IloCplex.and()`) take other goals as parameters, goals can be combined into aggregate goals. In fact, this is how goals are typically used for specifying a branching strategy. A typical return goal of a user-written `execute()` method for C++ looks like this:

```
return AndGoal(OrGoal(var <= IloFloor(val), var >= IloFloor(val)+1), this);
```

and for Java looks like this:

```
return cplex.and(cplex.or(cplex.leGoal(var, Math.floor(val)),
                          cplex.geGoal(var, Math.floor(val)+1)), this);
```

For the C++ case, note that since this statement would be called from the `execute()` method of a subclass of `IloCplex::GoalI`, the full method name `IloCplex::GoalI::OrGoal()` can be abbreviated to `OrGoal` (and likewise `AndGoal`).

This return statement returns an `And` goal that first executes the `Or` goal and then the current goal itself specified by the `this` parameter. When the `Or` goal is executed next, it will create two subnodes. In the first subnode, the first local cut goal representing $\texttt{var} \leq \lfloor \texttt{val} \rfloor$ (where $\lfloor \texttt{val} \rfloor$ denotes the floor of `val`) will be executed, thus adding the constraint $\texttt{var} \leq \lfloor \texttt{val} \rfloor$ for the subtree of this node. Similarly, the second subnode will be created, and when executing its constraint goal the constraint $\texttt{var} \geq \lfloor \texttt{val} \rfloor + 1$ will be added for the subtree. `this` is then executed on each of the nodes that have just been created; the same goal is used for both subtrees. Further details on how goals are processed will be discussed later.

We will discuss the execution of goals in the branch & cut search tree in more detail below. Before doing so, let us go through a first example. This example is available as file `ilogoalex1.cpp` in the `examples/src` subdirectory of your ILOG CPLEX distribution. The equivalent Java version can be found as `GoalEx1.java` in the same location.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilogoalex1.cpp
// Version 8.1
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilogoalex1.cpp - A simple goal example.
//
```

```
//                  Create a branch goal that chooses the variable
//                  with the largest objective from amongst those
//                  with the largest integer infeasibility.
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void usage (const char *progname);


// Branch on var with largest objective coefficient
// among those with largest infeasibility

ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars) {
   IloNumArray x;
   IloNumArray obj;
   IntegerFeasibilityArray feas;

   x    = IloNumArray(getEnv());
   obj  = IloNumArray(getEnv());
   feas = IntegerFeasibilityArray(getEnv());
   getValues(x, vars);
   getObjCoefs(obj, vars);
   getFeasibilities(feas, vars);

   IloInt bestj  = -1;
   IloNum maxinf = 0.0;
   IloNum maxobj = 0.0;
   IloInt cols = vars.getSize();
   for (IloInt j = 0; j < cols; ++j) {
      if ( feas[j] == Infeasible ) {
         IloNum xj_inf = x[j] - IloFloor (x[j]);
         if ( xj_inf > 0.5 )
            xj_inf = 1.0 - xj_inf;
         if ( xj_inf >= maxinf                              &&
             (xj_inf > maxinf || IloAbs (obj[j]) >= maxobj)  ) {
            bestj  = j;
            maxinf = xj_inf;
            maxobj = IloAbs (obj[j]);
         }
      }
   }

   IloCplex::Goal res;
   if ( bestj >= 0 ) {
      res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                           vars[bestj] <= IloFloor(x[bestj])),
                    this);
   }

   x.end();
```

```
   obj.end();
   feas.end();

   return res;
}


int
main (int argc, char **argv)
{
   IloEnv   env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);
      cplex.solve(MyBranchGoal(env, var));

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      cout << "Solution status = " << cplex.getStatus() << endl;
      cout << "Solution value  = " << cplex.getObjValue() << endl;
      cout << "Values          = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }

   env.end();

   return 0;
}


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
}
```

This example shows how to implement and use a goal for controlling the branch strategy used by IloCplex. As we discussed, goals are implemented as subclasses of class IloCplex::GoalI (IloCplex.Goal). In the C++ version of the example listed above we do so by using the macro

```
ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars)
```

instead. This macro defines two things, class MyBranchGoalI and function

```
IloCplex::Goal MyBranchGoal(IloEnv env, IloNumVarArray vars);
```

The class MyBranchGoalI is defined as a subclass of class IloCplex::GoalI (IloCplex.Goal) and has a private member IloNumVarArray vars. The function MyBranchGoal creates an instance of class MyBranchGoalI, initializes the member vars to the parameter vars passed to the function, and returns a handle to the new goal object. The braces "{ ... }" following the macro enclose the implementation of method MyBranchGoalI::execute() containing the actual code of the goal.

The use of the macro is very convenient as the amount of user code is equivalent to the amount for defining a function, but with a slightly unusual syntax. IloCplex provides 7 such macros that can be used for defining goals with 0 to 6 private members. If more than 6 members are needed, IloCplex::GoalI (IloCplex.Goal) must be subclassed *by hand*.

Since the Java programing language does not provide macros, a subclass of IloCplex.Goal must always be implemented by hand. In this example this class is called MyBranchGoal and we do not provide a helper function for creating an instance of that class as the macro does in the case of C++.

The goal is then used for solving the extracted node by calling:

```
cplex.solve(MyBranchGoal(env, var));
```

for C++, or for Java:

```
cplex.solve(new MyBranchGoal(var));
```

instead of the usual cplex.solve(). The rest of the main() function contains nothing new and will not be discussed any further.

In the implementation of the goal, or more precisely its method execute(), we start by declaring and initializing some arrays. These arrays are then used by methods of class IloCplex::GoalI (IloCplex.Goal) to query information about the node subproblem and the solution of its relaxation. Method getValues() is used to query the solution values for the variables in vars, method getObjCoefs() is used to query the linear objective function coefficients for these variables, and method getFeasibilities() is used to query feasibility statuses for them. The feasibility status of a variable indicates whether IloCplex considers the current solution value of the variable to be integer feasible or not. IloCplex::GoalI (IloCplex.Goal) provides a wealth of other query methods. For details, see the *ILOG CPLEX Reference Manual*.

Once we have gathered information about the variables, their objective coefficients, and their current feasibility statuses, we compute the index of an integer infeasible variable in `vars` that has the largest objective coefficients among the variables with largest integer infeasibility. That index is recorded in variable `bestj`.

Then we create a new goal handle object `res`. By default, this is initialized to an empty goal. However, if an integer infeasible variable was found among those in `vars` variable `bestj` will be ≥ `0` and we assign a non-empty goal to `res`:

```
res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                     vars[bestj] <= IloFloor(x[bestj])),
              this);
```

This goal creates two branches, one for `vars[bestj]` ≤ ⌊`x[bestj]`⌋ and one for `vars[bestj]` ≥ ⌊`x[bestj]`⌋ `+ 1` and continues branching in both subtrees with the same goal `this`. Finally, we call method `end()` for all temporary arrays and return goal `res`.

Since Java objects are garbage collected, there is no need for variable `res`. Instead, depending on the availability of an integer infeasible variable, the `null` goal is returned or the returned goal is created in the return statement itself:

```
return cplex.and(cplex.or(cplex.geGoal(_vars[bestj], Math.floor(x[bestj]))+1,
                          cplex.leGoal(_vars[bestj], Math.floor(x[bestj]))),
                 this);
```

The Goal Stack

To better understand how goals are executed, we need to understand the concept of the *goal stack*. Every node has its own goal stack. When calling `cplex.solve(goal)`, the goal stack of the root node is simply initialized with `goal` and then calls the regular `cplex.solve()` method.

When `IloCplex` processes a node, it pops the first goal from the node's goal stack and calls method `execute()`. If a nonempty goal is returned, it is simply pushed back on the stack. `IloCplex` keeps doing this until the node becomes inactive or the node's goal stack becomes empty. When the node stack is empty, `IloCplex` continues with its built-in search strategy for the subtree rooted at this node.

Let us now review the different types of goals in light of the goal stack:

1. As we already discussed, the `Or` goal creates child nodes. `IloCplex` first initializes the goal stack of every child node with a copy of the remaining goal stack of the current node. Then it pushes the goal passed as the parameter to the `Or` goal on the goal stack of the corresponding node. Finally, the current node is deactivated, and `IloCplex` continues search by picking a new active node from the tree to process.
2. The `And` goal simply pushes the goals passed as parameters onto the goal stack in reverse

order. Thus, when popping the goals from the stack for execution, they will be executed in the same order as they were passed as parameters to method And goal.

3. When a Fail goal executes, the current node is simply deactivated, and IloCplex continues on another active node from the tree. In other words, IloCplex discontinues its search below the current node.

4. When a local cut goal is executed, its constraints are added to the node problem as local cuts and the relaxation is re-solved.

5. An empty goal cannot be executed. Thus, empty goals are not pushed on the goal stack. If the goal stack is empty, IloCplex continues with the built-in branching strategy.

With this understanding, let us revisit what really goes on when a goal returns

```
return AndGoal(OrGoal(var <= IloFloor(val), var >= IloFloor(val)+1), this);
```

The And goal is pushed onto the current node's goal stack, only to be immediately popped back off of it. When it is executed, it will push this on the goal stack and then the Or goal. Thus, the Or goal is the next goal that IloCplex pops and executes. The Or goal creates two subnodes, and initializes their goal stacks with copies of the goal stack of the current node. At this point both subnodes will have this on top of their goal stacks. Next, the Or goal will push a local cut goal for $var \leq \lfloor val \rfloor$ (where $\lfloor val \rfloor$ denotes the floor of val) on the goal stack of the first subnode. Similarly, it pushes a local cut goal for $var \geq \lfloor val \rfloor + 1$ on the goal stack of the second subnode. Finally, the current node is deactivated and IloCplex continues its search with a new active node from the tree.

When IloCplex processes one of the subnodes that have been created by the Or goal, it will pop and execute the first goal from the node's goal stack. As we just saw, this will be a local cut goal. Thus IloCplex adds the constraint to the node problem and re-solves the relaxation. Next, this will be popped from the goal stack and executed. This means that the same search strategy as implemented in the original goal is applied at that node.

Memory Management and Goals

Java uses garbage collection to handle all memory management issues. Thus the following only applies to the C++ library. Java users may safely skip ahead to *Cuts and Goals*.

To conserve memory, IloCplex only stores active nodes of the tree and deletes nodes as soon as they become inactive. When deleting nodes, IloCplex also deletes the goal stacks associated with them, including all goals they may still contain. In other words, IloCplex takes over memory management for goals.

It does so by keeping track of how many references to every goal are in use. As soon as this number drops to zero (0), the goal is automatically deleted. This technique is known as reference counting.

IloCplex implements reference counting in the handle class IloCplex::Goal. Every IloCplex::GoalI object maintains a count of how many IloCplex::Goal handle

objects refer to it. The assignment operator, the constructors, and the destructor of class `IloCplex::Goal` are implemented in such a way as to keep the reference count up-to-date. This means that users should always access goals through handle objects, rather than keeping their own pointers to implementation objects.

Other than that, nothing special needs to be observed when dealing with goals. In particular, goals don't have `end()` methods like other handle classes in ILOG Concert Technology. Instead, `IloCplex` goal objects are automatically deleted when no more references to them exist.

Local cut goals contain `IloRange` objects. Since the `IloRange` object is only applied when the goal is executed, method `end()` must not be called for a range constraint from which a local cut goal is built. The goal will take over memory management for the constraints and call method `end()` when the goal itself is destroyed. Also, an `IloRange` object can only be used in exactly one local cut goal. Similarly, method `end()` must not be called for `IloRangeArray` objects that are passed to local cut goals. Also such arrays must not contain duplicate elements.

Going back to example `ilogoalex1.cpp`, we see that method `end()` is called for the temporary arrays `x`, `obj`, and `feas` at the end of the `execute()` method. Though a bit hidden, two `IloRange` constraints are constructed for the goal, corresponding to the parameters of the `Or` goal. `IloCplex` takes over memory management for these two constraints as soon as they are enclosed in a goal. This happens via the implicit constructor `IloCplex::Goal::Goal(IloRange rng)` that is called when the range constraints are passed as parameters to the `Or` goal.

In summary, the user is responsible for calling `end()` on all ILOG Concert technology object created in a goal, except if they have been passed as parameters to a new goal.

Also, user code in the `execute()` method is not allowed to modify existing ILOG Concert Technology objects in any way. `IloCplex` uses an optimized memory management system within goals for dealing with temporary objects. However, this memory management system cannot be mixed with the default memory management system used by ILOG Concert Technology. Thus, for example, it is illegal to add an element to array `vars` in the example, since this array has been created outside of the goal.

Cuts and Goals

Goals can also be used to add global cuts. Whereas local cuts are respected only in a subtree, global cuts are added to the entire problem and are therefore respected at every node after they have been added.

Global cuts can be added similarly to local cuts by using a global cut goal. A global cut goal is created with method `IloCplex::GoalI::GlobalCutGoal()` (`IloCplex.globalCutGoal()`). This method takes an `IloRange` or an `IloRangeArray` (`IloRange[]`) object as its

parameter and returns a goal. When the goal executes, it adds the constraints as global cuts to the problem.

Example `ilogoalex2.cpp` shows the use of `IloCplex::GoalI::GlobalCutGoal()` for solving the noswot MILP model. This is a relatively small model from the MIPLIB 3.0 test set, consisting only of 128 variables. Nonetheless, it is very hard to solve without adding special cuts.

Although it is now solvable directly, the computation time is in the order of several hours on state-of-the-art computers. However, cuts can be derived, and the addition of these cuts makes the problem solvable in a matter of minutes or seconds. These cuts are:

```
x21 - x22 <= 0
x22 - x23 <= 0
x23 - x24 <= 0
2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51 <= 20.25
2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52 <= 20.25
2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53 <= 20.25
2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54 <= 20.25
2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55 <= 16.25
```

These cuts have been derived after interpreting the model as a resource allocation model on five machines with scheduling, horizon constraints, and transaction times. The first three cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs.

Of course the best way to solve the `noswot` model with these cuts is to simply add them to the model before calling the optimizer. However, for demonstration purposes, we will add the cuts using a goal. The source code of this example can be found in the `examples/src` directory of the ILOG CPLEX distribution. The equivalent Java version can be found as `GoalEx2.java` in the same location.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilogoalex2.cpp
// Version 8.1
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilogoalex2.cpp -- Solve noswot by adding cuts through the goal API
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
```

```
ILOCPLEXGOAL3(CutGoal, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
   if ( isIntegerFeasible() )
      return 0;

   IloCplex::Goal goal = AndGoal(BranchAsCplexGoal(getEnv()), this);
   IloInt n = lhs.getSize();
   for (IloInt i = 0; i < n; ++i) {
      IloNum xrhs = rhs[i];
      if ( xrhs < IloInfinity  &&  getValue(lhs[i]) > xrhs + eps ) {
         goal = AndGoal(GlobalCutGoal(lhs[i] <= xrhs), goal);
         rhs[i] = IloInfinity;
      }
   }
   return goal;
}

void
makeCuts(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs)
{
   IloNumVar x11, x12, x13, x14, x15;
   IloNumVar x21, x22, x23, x24, x25;
   IloNumVar x31, x32, x33, x34, x35;
   IloNumVar x41, x42, x43, x44, x45;
   IloNumVar x51, x52, x53, x54, x55;
   IloNumVar w11, w12, w13, w14, w15;
   IloNumVar w21, w22, w23, w24, w25;
   IloNumVar w31, w32, w33, w34, w35;
   IloNumVar w41, w42, w43, w44, w45;
   IloNumVar w51, w52, w53, w54, w55;
   IloInt num = vars.getSize();

   for (IloInt i = 0; i < num; ++i) {
      if      ( strcmp(vars[i].getName(), "X11") == 0 ) x11 = vars[i];
      else if ( strcmp(vars[i].getName(), "X12") == 0 ) x12 = vars[i];
      else if ( strcmp(vars[i].getName(), "X13") == 0 ) x13 = vars[i];
      else if ( strcmp(vars[i].getName(), "X14") == 0 ) x14 = vars[i];
      else if ( strcmp(vars[i].getName(), "X15") == 0 ) x15 = vars[i];
      else if ( strcmp(vars[i].getName(), "X21") == 0 ) x21 = vars[i];
      else if ( strcmp(vars[i].getName(), "X22") == 0 ) x22 = vars[i];
      else if ( strcmp(vars[i].getName(), "X23") == 0 ) x23 = vars[i];
      else if ( strcmp(vars[i].getName(), "X24") == 0 ) x24 = vars[i];
      else if ( strcmp(vars[i].getName(), "X25") == 0 ) x25 = vars[i];
      else if ( strcmp(vars[i].getName(), "X31") == 0 ) x31 = vars[i];
      else if ( strcmp(vars[i].getName(), "X32") == 0 ) x32 = vars[i];
      else if ( strcmp(vars[i].getName(), "X33") == 0 ) x33 = vars[i];
      else if ( strcmp(vars[i].getName(), "X34") == 0 ) x34 = vars[i];
      else if ( strcmp(vars[i].getName(), "X35") == 0 ) x35 = vars[i];
      else if ( strcmp(vars[i].getName(), "X41") == 0 ) x41 = vars[i];
      else if ( strcmp(vars[i].getName(), "X42") == 0 ) x42 = vars[i];
      else if ( strcmp(vars[i].getName(), "X43") == 0 ) x43 = vars[i];
      else if ( strcmp(vars[i].getName(), "X44") == 0 ) x44 = vars[i];
```

```
   else if ( strcmp(vars[i].getName(), "X45") == 0 ) x45 = vars[i];
   else if ( strcmp(vars[i].getName(), "X51") == 0 ) x51 = vars[i];
   else if ( strcmp(vars[i].getName(), "X52") == 0 ) x52 = vars[i];
   else if ( strcmp(vars[i].getName(), "X53") == 0 ) x53 = vars[i];
   else if ( strcmp(vars[i].getName(), "X54") == 0 ) x54 = vars[i];
   else if ( strcmp(vars[i].getName(), "X55") == 0 ) x55 = vars[i];
   else if ( strcmp(vars[i].getName(), "W11") == 0 ) w11 = vars[i];
   else if ( strcmp(vars[i].getName(), "W12") == 0 ) w12 = vars[i];
   else if ( strcmp(vars[i].getName(), "W13") == 0 ) w13 = vars[i];
   else if ( strcmp(vars[i].getName(), "W14") == 0 ) w14 = vars[i];
   else if ( strcmp(vars[i].getName(), "W15") == 0 ) w15 = vars[i];
   else if ( strcmp(vars[i].getName(), "W21") == 0 ) w21 = vars[i];
   else if ( strcmp(vars[i].getName(), "W22") == 0 ) w22 = vars[i];
   else if ( strcmp(vars[i].getName(), "W23") == 0 ) w23 = vars[i];
   else if ( strcmp(vars[i].getName(), "W24") == 0 ) w24 = vars[i];
   else if ( strcmp(vars[i].getName(), "W25") == 0 ) w25 = vars[i];
   else if ( strcmp(vars[i].getName(), "W31") == 0 ) w31 = vars[i];
   else if ( strcmp(vars[i].getName(), "W32") == 0 ) w32 = vars[i];
   else if ( strcmp(vars[i].getName(), "W33") == 0 ) w33 = vars[i];
   else if ( strcmp(vars[i].getName(), "W34") == 0 ) w34 = vars[i];
   else if ( strcmp(vars[i].getName(), "W35") == 0 ) w35 = vars[i];
   else if ( strcmp(vars[i].getName(), "W41") == 0 ) w41 = vars[i];
   else if ( strcmp(vars[i].getName(), "W42") == 0 ) w42 = vars[i];
   else if ( strcmp(vars[i].getName(), "W43") == 0 ) w43 = vars[i];
   else if ( strcmp(vars[i].getName(), "W44") == 0 ) w44 = vars[i];
   else if ( strcmp(vars[i].getName(), "W45") == 0 ) w45 = vars[i];
   else if ( strcmp(vars[i].getName(), "W51") == 0 ) w51 = vars[i];
   else if ( strcmp(vars[i].getName(), "W52") == 0 ) w52 = vars[i];
   else if ( strcmp(vars[i].getName(), "W53") == 0 ) w53 = vars[i];
   else if ( strcmp(vars[i].getName(), "W54") == 0 ) w54 = vars[i];
   else if ( strcmp(vars[i].getName(), "W55") == 0 ) w55 = vars[i];
}

lhs.add(x21 - x22);
rhs.add(0.0);

lhs.add(x22 - x23);
rhs.add(0.0);

lhs.add(x23 - x24);
rhs.add(0.0);

lhs.add(2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
        0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51);
rhs.add(20.25);

lhs.add(2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
        0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52);
rhs.add(20.25);

lhs.add(2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
        0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53);
rhs.add(20.25);
```

```
   lhs.add(2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
           0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54);
   rhs.add(20.25);

   lhs.add(2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
           0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55);
   rhs.add(16.25);
}


int
main(int argc, char** argv)
{
   IloEnv env;
   try {
      IloModel m;
      IloCplex cplex(env);

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  con(env);

      env.out() << "reading ../../../examples/data/noswot.mps" << endl;
      cplex.importModel(m, "../../../examples/data/noswot.mps", obj, var, con);

      env.out() << "extracting model ..." << endl;
      cplex.extract(m);
      cplex.setParam(IloCplex::MIPInterval, 1000);

      env.out() << "initializing cuts ..." << endl;
      IloExprArray lhs(env);
      IloNumArray  rhs(env);
      makeCuts(var, lhs, rhs);

      env.out() << "solving model ...\n";
      cplex.solve(CutGoal(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
      env.out() << "solution status is " << cplex.getStatus() << endl;
      env.out() << "solution value  is " << cplex.getObjValue() << endl;
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   env.end();
   return 0;
}
```

The goal `CutGoal` in this example receives a list of "less than" constraints to use as global cuts and a tolerance value `eps`. The constraints are passed to the goal as an array of `lhs` expressions and an array of corresponding `rhs` values. Both are initialized in function `makeCuts`.

The goal `CutGoal` checks if any of the constraints passed to it are violated by more than the tolerance value. It adds violated constraints as global cuts. Other than that, it follows the branching strategy `IloCplex` would use on its own.

The goal starts out by checking if the solution of the continuous relaxation of the current node subproblem is integer feasible. This is done by calling method `isIntegerFeasible()`. If the current solution is integer feasible, a candidate for a new incumbent has been found and the goal returns the empty goal to instruct `IloCplex` to continue on its own.

Otherwise, the goal checks if any of the constraints passed to it are violated. It computes the value of every `lhs` expression for current solution by calling `getValue(lhs[i])`. The result is compared against the corresponding `rhs` value `rhs[i]`. If a violation of more than `eps` is detected, the constraint is added as a global cut and the `rhs` value will be set to `IloInfinity` to avoid checking it again unnecessarily.

The global cut goal for `lhs[i]` $\leq$`rhs[i]` is created by calling method `GlobalCutGoal`. It is then combined with goal `goal` using method `AndGoal`, so that the new global cut goal will be executed first. The resulting goal is stored again in variable `goal`. Before adding any global cut goals, variable `goal` is initialized as

```
IloCplex::Goal goal = AndGoal(BranchAsCplexGoal(getEnv()), this);
```

for C++, or for Java:

```
cplex.and(cplex.branchAsCplex(), this);
```

The method `BranchAsCplexGoal(getEnv())` (`(cplex.branchAsCplex())`) creates a goal that branches in the same way as the built-in branch procedure. By adding this goal, the current goal will be executed for the entire subtree.

Thus the goal returned by `CutGoal` will add all currently violated constraints as global cuts one by one. Then it will branch in the way `IloCplex` would branch without any goals and execute the `CutGoal` again in the child nodes.

Injecting Solutions

At any time in the execution of a goal, you may find that, for example, by slightly manipulating the current node subproblem solution, you may construct a solution to your model. Such solutions are called *heuristic solutions*, and a procedure that generates them is called a *heuristic*.

Heuristic solutions can be injected into the branch & cut search by creating a solution goal with method `IloCplex::GoalI::SolutionGoal()` (`IloCplex.solutionGoal()`). Such a goal can be returned typically as a subgoal of `And` goal much like global cut goals.

When `IloCplex` executes a solution goal it does not immediately use the specified solution as a potential new incumbent. The reason is that with goals, part of the model may be

specified via global cuts or through specialized branching strategies. Thus the solution needs first to be tested for feasibility with respect to the entire model including any part of the model specified through goals.

To test if an injected solution is feasible with all goals the `execute()` method of a solution goal `IloCplex` first creates a subnode of the current node. This subnode will of course inherit the goal stack from its parent. In addition the solution goal will push local cuts onto the stack of the subnode such that all variables are fixed to the values of the injected solution.

By processing this subnode (as the next node) `IloCplex` ensures that the solution is feasible with respect to all goals, or is otherwise discarded. Goals that have been executed so far are either reflected as global cuts or by the local cuts that are active at the current node. Thus, if the relaxation remains feasible after the variable fixings have been added, the feasibility of these goals is assured.

If at that point the goal stack is not empty, the goals on the goal stack need to be checked for feasibility as well. Thus by continuing to execute the goals from the goal stack `IloCplex` will either prove feasibility of the solution with respect to the remaining goals or, in case the relaxation becomes infeasible, determine it to be really infeasible and discard the solution. The rest of the branch & cut search remain unaffected by all of this.

The benefit of this approach is that your heuristic need not be aware of the entire model including all its parts that might be implemented via goals. Your heuristic can still safely be used, as `IloCplex` will ensure feasibility for the entire model. However, there are some performance considerations to observe. If parts of the model specified with goals are dominant, heuristic solutions you generate might need to be rejected so frequently that you do not get enough payoff for the work of running the heuristic. Also, your heuristic should account for the global and local cuts that have been added at the node where you run your heuristic so that a solution candidate is not rejected right away and the work wasted.

Controlling Goal-Controlled Search

So far, we have discussed how to control the branching and cut generation of `IloCplex` branch & cut search. The remaining missing piece is the node selection strategy. The node selection strategy determines which of the active nodes in the tree `IloCplex` chooses when it selects the next node for processing. `IloCplex` has several built-in node selection strategies, selected through parameter `NodeSel`.

When using goal-controlled search, node evaluators are used to override the built-in node selection strategy. The process is as follows. You combine a goal with a node evaluator by calling method `IloCplex::Goal::Apply()` (`IloCplex.apply()`). This method returns a new goal that implements the same search strategy as the goal passed as the parameter, but adds the node evaluator to every node in the subtree defined by the goal. Therefore, nodes may have a list of evaluators attached to them.

When node evaluators are used, nodes are selected as follows:

- `IloCplex` starts to choose the node with the built-in strategy as a first candidate.
- Then `IloCplex` loops over all remaining active nodes and considers choosing them instead.
- If a node has the same evaluator attached to it as the current candidate, the evaluator is asked if this node should take precedence over the current candidate. If the response is positive, the node under investigation becomes the new candidate and the test against other nodes continues.

If a node has multiple evaluators attached, they are consulted in the order the evaluators have been applied. This occurs as follows:

- If the first evaluator prefers one node over the other, the preferred node is used as candidate and the next node is considered.
- If the first evaluator does not give preference to one node over the other, the second evaluator is considered, and so on.

Thus, by adding multiple evaluators, you can build composite node selection strategies where later evaluators are used for breaking ties in previous ones.

Node evaluators are implemented as subclasses of class `IloCplex::NodeEvaluatorI`. Class `IloCplex::NodeEvaluator` is the handle class for node evaluators. In Java, node evaluators are implemented in objects of type `IloCplex.NodeEvaluator` (and there are no handle classes). Like goals, node evaluators use reference counting for memory management. As a result, you should always use the handle objects when dealing with node evaluators, and there is no method `end()` to be called.

Node evaluators use a two-step process to decide whether one node should take precedence over another. First, the evaluator computes a value for every node to which it is attached. This is done by calling method

```
IloNum IloCplex::NodeEvaluatorI::evaluate();
```

in the case of C++, and for Java by calling method:

```
double IloCplex.NodeEvaluator.evaluate();
```

This method must be implemented by users who write their own node evaluators. In method `evaluate()`, the protected member functions of class `IloCplex::NodeEvaluatorI` (`IloCplex.NodeEvaluator`) can be called to query information about the node being evaluated. The method `evaluate()` must compute and return an evaluation value that is used later on in the second step to compare two nodes and select one of them. The `evaluate()` method is called only once for every node, and the result is cached and reused whenever the node is compared against another node with the evaluator.

The second step consists of comparing the current candidate to another node. This only happens for evaluators that are shared by the current candidate and the other node. By

default, the candidate is replaced by the other node if its evaluation value is smaller than that of the candidate. This behavior can be altered by overwriting the method

```
IloBool IloCplex::NodeEvaluatorI::subsume(IloNum candVal, IloNum nodeVal);
```

or, in the case of Java:

```
boolean IloCplex.NodeEvaluator.subsume(double candVal, double nodeVal);
```

`IloCplex` calls this method of an evaluator attached to the current candidate if the node being compared also has the same evaluator attached. The first parameter `candVal` is the evaluation value the evaluator has previously computed for the current candidate, and `nodeVal` is the evaluation value the evaluator has previously computed for the node being tested. If this method returns `IloTrue` (`true`), the candidate is replaced. Otherwise the method is called again with reversed parameters. If it still returns `IloFalse` (`false`), both nodes are tied with respect to that evaluator, and the next evaluator they share is consulted. Otherwise, the current candidate is kept and tested against the next node.

There are two more virtual methods defined for node evaluators that should be considered when implementing your own node evaluator. Method `void init()` is called right before `evaluate()` is called for the first time, thus allowing you to initialize internal data of the evaluator. When this happens, the evaluator has been initialized to the first node to be evaluated, thus information about this node can be queried by calling the methods of class `IloCplex::NodeEvaluatorI` (`IloCplex.NodeEvaluator`).

Finally, for C++ method

```
IloCplex::NodeEvaluatorI* IloCplex::NodeEvaluatorI::duplicateEvaluator();
```

must be implemented by the user to return a copy of the invoking node evaluator object. This method is called by `IloCplex` to create copies of the evaluator for parallel branch & cut search.

Example `ilogoalex3.cpp` shows how to use node evaluators to implement a node selection strategy that chooses the deepest active node in tree among those with maximal sum of integer infeasibilities. Example `ilogoalex3.cpp` can be found in the `examples/src` directory of your distribution. The equivalent Java version can be found as file `Goalex3.java` at the same location.

```
// ------------------------------------------------------------ -*- C++ -*-
// File: examples/src/ilogoalex3.cpp
// Version 8.1
// ---------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ---------------------------------------------------------------------------
//
// ilogoalex3.cpp - Node selectors in goal search
```

```
//
//                  This is an extension of example ilogoalex1.cpp
//                  It adds node evaluators to further control the
//                  goal based search.  The node evaluator chooses
//                  the node that is deepest in the tree among those
//                  with maximum sum of integer infeasibilities
//

#include <ilcplex/ilocplex.h>

ILOSTLBEGIN

static void usage (const char *progname);


// Branch on var with largest objective coefficient
// among those with largest infeasibility

ILOCPLEXGOAL1(MyBranchGoal, IloNumVarArray, vars) {
   IloNumArray x;
   IloNumArray obj;
   IntegerFeasibilityArray feas;

   x    = IloNumArray(getEnv());
   obj  = IloNumArray(getEnv());
   feas = IntegerFeasibilityArray(getEnv());
   getValues(x, vars);
   getObjCoefs(obj, vars);
   getFeasibilities(feas, vars);

   IloInt bestj  = -1;
   IloNum maxinf = 0.0;
   IloNum maxobj = 0.0;
   IloInt cols = vars.getSize();
   for (IloInt j = 0; j < cols; ++j) {
      if ( feas[j] == Infeasible ) {
         IloNum xj_inf = x[j] - IloFloor (x[j]);
         if ( xj_inf > 0.5 )  xj_inf = 1.0 - xj_inf;
         if ( xj_inf >= maxinf                                &&
             (xj_inf > maxinf || IloAbs (obj[j]) >= maxobj)  ) {
            bestj  = j;
            maxinf = xj_inf;
            maxobj = IloAbs (obj[j]);
         }
      }
   }

   IloCplex::Goal res;
   if ( bestj >= 0 ) {
      res = AndGoal(OrGoal(vars[bestj] >= IloFloor(x[bestj])+1,
                           vars[bestj] <= IloFloor(x[bestj])),
                    this);
   }
```

```
   x.end();
   obj.end();
   feas.end();

   return res;
}


// Depth first search node evaluator

class DepthEvaluatorI : public IloCplex::NodeEvaluatorI {
public:
   IloNum evaluate() const { return -getDepth(); }

   IloCplex::NodeEvaluatorI *duplicateEvaluator() {
      return new DepthEvaluatorI();
   }
};

IloCplex::NodeEvaluator DepthEvaluator() {
   return new DepthEvaluatorI();
}


// integer infeasibility sum node evaluator

class IISumEvaluatorI : public IloCplex::NodeEvaluatorI {
public:
   IloNum evaluate() const { return -getInfeasibilitySum(); }

   IloCplex::NodeEvaluatorI *duplicateEvaluator() {
      return new IISumEvaluatorI();
   }
};

IloCplex::NodeEvaluator IISumEvaluator() {
   return new IISumEvaluatorI();
}


int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env);
      IloCplex cplex(env);

      if ( argc != 2 ) {
         usage (argv[0]);
         throw(-1);
```

```
      }

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      cplex.importModel(model, argv[1], obj, var, rng);

      cplex.extract(model);

      IloCplex::Goal iiSumGoal = IloCplex::Apply(cplex,
                                                 MyBranchGoal(env, var),
                                                 IISumEvaluator());
      IloCplex::Goal depthGoal = IloCplex::Apply(cplex,
                                                 iiSumGoal,
                                                 DepthEvaluator());
      cplex.solve(depthGoal);

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      cout << "Solution status = " << cplex.getStatus() << endl;
      cout << "Solution value  = " << cplex.getObjValue() << endl;
      cout << "Values          = " << vals << endl;
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }

   env.end();

   return 0;
}


static void usage (const char *progname)
{
   cerr << "Usage: " << progname << " filename" << endl;
   cerr << "   where filename is a file with extension " << endl;
   cerr << "      MPS, SAV, or LP (lower case is allowed)" << endl;
   cerr << " Exiting..." << endl;
}
```

As this example is an extension of example `ilogoalex1.cpp`, we shall only concentrate on the differences. Also, we will discuss the example only in terms of the C++ version; the Java version has identical structure and design and differs only in syntax.

The first is the definition of class `DepthEvaluatorI` as a subclass of `IloCplex::NodeEvaluatorI`. It implement methods `evaluate()` and `duplicateEvaluator()`. Method `evaluate()` simply returns the negative depth value queried for the current node by calling method `getDepth()`. Since `IloCplex` by default chooses nodes with the lowest evaluation value, this evaluator will favor nodes deep in the tree. Method `duplicateEvaluator()` simply returns a copy of the invoking object by

calling the (default) copy constructor. Along with the class, we also define function `DepthEvaluator()` which creates an instance of class `DepthEvaluatorI` and returns a handle to it.

Similarly, we define class `IISumEvaluatorI` and function `IISumEvaluator()`. The `evaluate()` method returns the negation of the sum of integer infeasibilities of the node being evaluated. This number is obtained by calling method `getInfeasibilitySum()`. Thus, this evaluator favors nodes with larger sums of integer infeasibilities.

This example uses the same search strategy as `ilogoalex1.cpp`, implemented in goal `MyBranchGoal`. However, it applies first the `IISumEvaluator` to select nodes with high integer infeasibility sum, to choose between nodes with the same integer infeasibility sum it applies the `DepthEvaluator`. Applying the `IISumEvaluator` is done with

```
IloCplex::Goal iiSumGoal = IloCplex::Apply(cplex,
                                           MyBranchGoal(env, var),
                                           IISumEvaluator());
```

The goal created by calling `MyBranchGoal` is merged with the evaluator created by calling `IISumEvaluator` into a new goal `iiSumGoal`. Similarly, the `iiSumGoal` is merged with the node evaluator created by calling `DepthEvaluator` into a new goal `depthGoal`:

```
IloCplex::Goal depthGoal = IloCplex::Apply(cplex,
                                           iiSumGoal,
                                           DepthEvaluator());
```

Thus, `depthGoal` represents a goal implementing the branching strategy defined by `MyBranchGoal`, but using `IISumEvaluator` as a primary node selection strategy and `DepthEvaluator` as a secondary node selection strategy for breaking ties. This goal is finally used for the branch & cut search by passing it to the `solve()` method.

Node evaluators are only active while the search is controlled by goals. That is, if the goal stack becomes empty at a node and `IloCplex` continues searching with its built-in search strategy, that search is no longer controlled by any node evaluator. In order to maintain control over the node selection strategy while using the `IloCplex` branch strategy, you can use the goal returned by method `IloCplex::GoalI::BranchAsCplexGoal()` (`IloCplex.branchAsCplex()`). A goal that follows the branching performed by `IloCplex` built-in strategy can be easily implemented as:

```
ILOCPLEXGOAL0(DefaultSearchGoal) {
  if ( !isIntegerFeasible() )
    return AndGoal(BranchAsCplexGoal(getEnv()), this);
  return 0;
}
```

Notice the test for integer feasibility. Without that test, we would create an endless loop because when an integer feasible solution has been found, `BranchAsCplex` goal does not change the node at all, and `this` would continue to be executed indefinitely.

Search Limits

As with node evaluators, it is possible to apply search limits to the branch & cut search controlled by goals. Search limits allow you to limit the search in certain subtrees; that is, they allow you to discontinue processing nodes when some condition applies. Search limits are implemented in subclasses of class `IloCplex::SearchLimitI` (`IloCplex.SearchLimit`), and the procedure for implementing and using them is very similar to that for node evaluators. See the reference manual for more details on implementing and using search limits.

Using Callbacks

This section introduces the topic of callback routines, which allow you to closely monitor and guide the behavior of ILOG CPLEX optimizers. It includes information on:

- Diagnostic Callbacks
- Control Callbacks for `IloCplex`

ILOG CPLEX callbacks allow user code to be executed regularly during an optimization. There are two types of callbacks, diagnostic callbacks and control callbacks, which are discussed separately in the following sections. To use callbacks with ILOG CPLEX, you must first write the callback function, and then pass it to ILOG CPLEX.

Diagnostic Callbacks

Diagnostic callbacks allow you to monitor an ongoing optimization, and optionally abort it. These callbacks are distinguished by the place where they are called during an optimization. There are 9 such places where diagnostic callbacks are called:

- The presolve callback is called regularly during presolve.
- The crossover callback is called regularly during crossover from a barrier solution to a simplex basis.
- The network callback is called regularly during the network simplex.
- The barrier callback is called at each iteration during the barrier algorithm.
- The simplex callback is called at each iteration during the simplex algorithm.
- The MIP callback is called at each node during the branch & cut search.
- The probing callback is called regularly during probing.
- The fractional cut callback is called regularly during the generation of fractional cuts.
- The disjunctive cut callback is called regularly during the generation of disjunctive cuts.

Implementing Callbacks In ILOG CPLEX with Concert Technology

Callbacks are accessed via the `IloCplex::Callback` handle class in the C++ version of `IloCplex`. It points to an implementation object of a subclass of `IloCplex::CallbackI`. In java, there is no handle class and one only deals with implementation classes which are subclasses of `IloCplex.Callback`. One such implementation class is provided for each type of callback. The implementation class provides the functions that can be used for the particular callback as protected member functions. To reflect the fact that some callbacks share part of their protected API, the callback classes are organized in a class hierarchy as shown by this diagram for C++:

```
IloCplex::CallbackI
   |
   +--- IloCplex::PresolveCallbackI
   |
   +--- IloCplex::CrossoverCallbackI
   |
   +--- IloCplex::NetworkCallbackI
   |
   +--- IloCplex::ContinuousCallbackI
   |       |
   |       +--- IloCplex::BarrierCallbackI
   |       |
   |       +--- IloCplex::SimplexCallbackI
   |
   +--- IloCplex::MIPCallbackI
           |
           +--- IloCplex::ProbingCallback
           |
           +--- IloCplex::FractionalCutCallbackI
           |
           +--- IloCplex::DisjunctiveCutCallbackI
```

The callback class hierarchy for Java is exactly the same, but the class names differ, in that there is no '`I`' at the end. For example the corresponding Java implementation class for `IloCplex::ContinuousCallbackI` is `IloCplex.ContinuousCallback`.

This means that, for example, all functions available for the MIP callback are also available for the probing, fractional cut, and disjunctive cut callbacks. In particular, the function to abort the current optimization is provided by the class `IloCplex::CallbackI` (`IloCplex.Callback`) and is thus available to all callbacks.

There are two ways of implementing callbacks for `IloCplex`: a more complex way that exposes all the C++ implementation details, and a simplified way that uses macros to handle the C++ technicalities. Since the Java programing language does not provide macros, only the more complex way is available for Java. We will first explain the more complex way and discuss the underlying design. To quickly implement your callback without details about the internal design, proceed directly to *Writing Callbacks with Macros* on page 343.

Writing Callback Classes by Hand

To implement your own callback for `IloCplex`, first select the callback class corresponding to the callback you want implemented. From it derive your own implementation class and overwrite the virtual method `main()`. This is where you implement the callback actions, using the protected member functions of the callback class from which you derived your callback or one of its base classes.

Next write a function that creates a new object of your implementation class using the environment operator new and returning it as an IloCplex::Callback handle object. Here is an example implementation of such a function:

```
IloCplex::Callback MyCallback(IloEnv env, IloInt num) {
                                  return (new (env) MyCallbackI(num));
}
```

It is not customary to write such a function for Java, but new is called explicitly for creating a callback object when needed. Once an implementation object of your callback is created (either with the constructor function in C++ or by directly calling the new operator for Java), use it with IloCplex by calling cplex.use() with the callback object as parameter. In C++, to remove a callback that is being used by a cplex object, call callback.end() on the IloCplex::Callback handle callback. In java there is no way of removing individual callbacks from your IloCplex object. Instead, you can remove all callbacks by calling cplex.clearCallbacks(). Since Java uses garbage collection for memory management, there is nothing equivalent to the end() method for callbacks in the Java API.

One object of a callback implementation class can be used with only one IloCplex object at a time. Thus, when you use a callback with more than one cplex object, a copy of the implementation object is created every time cplex.use() is called except for the first time. In C++, the method IloCplex::use() returns a handle to the callback object that has actually been installed to enable calling end() on it.

To construct the copies of the callback objects in C++, class IloCplex::CallbackI defines another pure virtual method:

```
virtual IloCplex::CallbackI* IloCplex::CallbackI::duplicateCallback()
     const = 0;
```

which must be implemented for your callback class. This method will be called to create the copies needed for using a callback on different cplex objects or on one cplex object with a parallel optimizer.

In most cases you can avoid writing callback classes by hand, using supplied macros that make the process as easy as implementing a function. You must implement a callback by hand only if the callback manages internal data not passed as arguments, or if the callback requires eight or more parameters.

Writing Callbacks with Macros

This is how to implement a callback using macros. Since macros are not supported in java, this technique will only apply to C++.

Start by determining which callback you want to implement and how many arguments to pass to the callback function. These two pieces of information determine the macro you need to use.

For example, to implement a simplex callback with one parameter, the macro is ILOSIMPLEXCALLBACK1. Generally, for every callback type XXX and any number of

parameters n from 0 to 7, there is a macro called ILOXXXCALLBACKn. The following table lists the callbacks and the corresponding macros and classes (where n is a placeholder for 0...7):

Table 9.4 *Callback Macros*

Callback	Macro	Class
presolve	ILOPRESOLVECALLBACKn	IloCplex::PresolveCallbackI
continuous	ILOCONTINUOUSCALLBACKn	IloCplex::ContinuousCallbackI
simplex	ILOSIMPLEXCALLBACKn	IloCplex::SimplexCallbackI
primal simplex	ILOPRIMALSIMPLEXCALLBACKn	IloCplex::SimplexCallbackI
dual simplex	ILODUALSIMPLEXCALLBACKn	IloCplex::SimplexCallbackI
barrier	ILOBARRIERCALLBACKn	IloCplex::BarrierCallbackI
crossover	ILOCROSSOVERCALLBACKn	IloCplex::CrossoverCallbackI
network	ILONETWORKCALLBACKn	IloCplex::NetworkCallbackI
MIP	ILOMIPCALLBACKn	IloCplex::MIPCallbackI
probing	ILOPROBINGCALLBACKn	IloCplex::ProbingCallbackI
fractional cut	ILOFRACTIONALCUTCALLBACKn	IloCplex::FractionalCutCallbackI
disjunctive cut	ILODISJUNCTIVECUTCALLBACKn	IloCplex::DisjunctiveCutCallbackI

The protected member functions of the corresponding class and its base classes determine the functions that can be called for implementing your callback (see the *ILOG CPLEX Reference Manual*).

Here is an example of how to implement a simplex callback with the name MyCallback that takes one parameter:

```
ILOSIMPLEXCALLBACK1(MyCallback, IloInt, num) {
  if ( getNiterations() == num ) abort();
}
```

This callback aborts the simplex algorithm at the numth iteration. It queries the current iteration number by calling function getNiterations(), which is a protected member function of class IloCplex::ContinuousCallbackI.

To use this callback with an `IloCplex` object `cplex`, simply call:

```
IloCplex::Callback mycallback = cplex.use(MyCallback(env, 10));
```

The callback that is added to `cplex` is returned by the method `use` and stored in variable `mycallback`. This allows you to call `mycallback.end()`to remove the callback from `cplex`. If you do not intend accessing your callback, for example in order to delete it before ending the environment, you may safely leave out the declaration and initialization of variable `mycallback`.

Callback Interface

Two callback classes in the hierarchy need extra attention. The first is the base class `IloCplex::CallbackI` (`IloCplex.CallbackI`). Since there is no corresponding callback in the Callable Library, this class cannot be used for implementing user callbacks. Instead, its purpose is to provide an interface common to all callback functions. This consists of the methods `getModel()`, which returns the model that is extracted to the `IloCplex` object that is calling the callback, `getEnv()`, which returns the corresponding environment (C++ only), and `abort()`, which aborts the current optimization. Further, methods `getNrows()` and `getNcols()` allow you to query the number of rows and columns of the current `cplex` LP matrix. These methods can be called from all callbacks.

> ***Note:*** *For C++ users, no manipulation of the model or, more precisely, any extracted modeling object is allowed during the execution of a callback. No modification is allowed of any array or expression not local to the callback function itself (that is, constructed and* `end()`*ed in it). The only exception is the modification of array elements. For example,* `x[i] = 0` *would be permissible, whereas* `x.add(0)` *would not unless* `x` *is a local array of the callback.*

The Continuous Callback

The second special callback class is `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback`). If you create a Continuous callback and use it with an `IloCplex` object, it will be used for both the barrier and the simplex callback. In other words, implementing and using one Continuous callback is equivalent to writing and using these two callbacks independently.

Example: Deriving the Simplex Callback

Example `ilolpex4.cpp` demonstrates the use of the simplex callback to print logging information at each iteration. It is a modification of example `ilolpex1.cpp`, so we will restrict our discussion to the differences. The following code:

```
ILOSIMPLEXCALLBACK0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
    if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
    cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

defines the callback `MyCallback` without parameters with the code enclosed in the outer `{}`. In Java the same callback is defined as:

```
static class MyCallback extends IloCplex.ContinuousCallback {
   public void main() throws IloException {
      System.out.print("Iteration " + getNiterations() + ": ");
      if ( isFeasible() )
         System.out.println("Objective = " + getObjValue());
      else
         System.out.println("Infeasibility measure = " +
getInfeasibility());
   }
}
```

The callback prints the iteration number. Then, depending on whether the current solution is feasible or not, it prints the objective value or infeasibility measure. The functions `getNiterations()`, `isFeasible()`, `getObjValue()`, and `getInfeasibility()` are member functions provided in the callback's base class `IloCplex::ContinuousCallbackI` (`IloCplex.ContinuousCallback`). See the *ILOG CPLEX Reference Manual* for the complete list of methods provided for each callback class.

Here is how the macro ILOSIMPLEXCALLBACK0 is expanded:

```
class MyCallbackI : public IloCplex::SimplexCallbackI {
  void main();
  IloCplex::CallbackI* duplicateCallback() const {
    return (new (getEnv()) MyCallbackI(*this));
  }
};
IloCplex::Callback MyCallback(IloEnv env) {
  return (IloCplex::Callback(new (env) MyCallbackI()));
}
void MyCallbackI::main() {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
    cout << "Objective = " << getObjValue() << endl;
  }
  else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}
```

The 0 in the macro indicates that 0 parameters are passed to the constructor of the callback. For callbacks requiring up to 7 parameters similar macros are defined where the 0 is replaced by the number of parameters, ranging from 1 through 7. For an example of this using the cut callback, see *Example: Controlling Cuts* on page 360. If you need more than 7 parameters, you will need to derive your callback class yourself without the help of a macro.

After the callback MyCallback is defined, it can be used with the line:

```
cplex.use(MyCallback(env));
```

in C++ or

```
cplex.use(new MyCallback());
```

in Java.

In the case of C++, function MyCallback creates an instance of the implementation class MyCallbackI. A handle to this implementation object is passed to cplex method use().

If your application defines more than one primal simplex callback object (possibly with different subclasses), only the last one passed to ILOG CPLEX with the use method is actually used during primal simplex. On the other hand, IloCplex can handle one callback for each callback class at the same time. For example a simplex callback and a MIP callback can be used at the same time.

Complete Program: ilolpex4.cpp

The complete program, ilolpex4.cpp, appears here and online in the standard distribution.

```
// -------------------------------------------------------------- -*- C++ -*-
// File: examples/src/ilolpex4.cpp
// Version 8.1
// -------------------------------------------------------------------------
```

```
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// ------------------------------------------------------------------------
//
// ilolpex4.cpp - Illustrating the CPLEX callback functionality.
//
// This is a modification of ilolpex1.c, where we use a callback
// function to print the iteration info, rather than have CPLEX
// do it.

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN


ILOSIMPLEXCALLBACK0(MyCallback) {
  cout << "Iteration " << getNiterations() << ": ";
  if ( isFeasible() ) {
     cout << "Objective = " << getObjValue() << endl;
  } else {
     cout << "Infeasibility measure = " << getInfeasibility() << endl;
  }
}


static void
   populatebycolumn (IloModel model, IloNumVarArray var, IloRangeArray rng);

int
main (int argc, char **argv)
{
   IloEnv env;
   try {
      IloModel model(env, "example");

      IloNumVarArray var(env);
      IloRangeArray  rng(env);
      populatebycolumn (model, var, rng);

      IloCplex cplex(model);
      cplex.setOut(env.getNullStream());
      cplex.setParam(IloCplex::RootAlg, IloCplex::Primal);
      cplex.use(MyCallback(env));
      cplex.solve();

      env.out() << "Solution status = " << cplex.getStatus() << endl;
      env.out() << "Solution value  = " << cplex.getObjValue() << endl;

      IloNumArray vals(env);
      cplex.getValues(vals, var);
      env.out() << "Values        = " << vals << endl;
      cplex.getSlacks(vals, rng);
```

```
      env.out() << "Slacks        = " << vals << endl;
      cplex.getDuals(vals, rng);
      env.out() << "Duals         = " << vals << endl;
      cplex.getReducedCosts(vals, var);
      env.out() << "Reduced Costs = " << vals << endl;

      cplex.exportModel("lpex4.lp");
   }
   catch (IloException& e) {
      cerr << "Concert exception caught: " << e << endl;
   }
   catch (...) {
      cerr << "Unknown exception caught" << endl;
   }

   env.end();
   return 0;
}  // END main


// To populate by column, we first create the rows, and then add the
// columns.

static void
populatebycolumn (IloModel model, IloNumVarArray x, IloRangeArray c)
{
   IloEnv env = model.getEnv();

   IloObjective obj = IloMaximize(env);
   c.add(IloRange(env, -IloInfinity, 20.0));
   c.add(IloRange(env, -IloInfinity, 30.0));

   x.add(IloNumVar(obj(1.0) + c[0](-1.0) + c[1]( 1.0), 0.0, 40.0));
   x.add(obj(2.0) + c[0]( 1.0) + c[1](-3.0));
   x.add(obj(3.0) + c[0]( 1.0) + c[1]( 1.0));

   model.add(obj);
   model.add(c);
}  // END populatebycolumn
```

Implementing Callbacks in the Callable Library

ILOG CPLEX optimization routines in the Callable Library incorporate a callback facility to allow your application to transfer control temporarily from ILOG CPLEX to the calling application. Using callbacks, your application can implement interrupt capability, for example, or create displays of optimization progress. Once control is transferred back to a function in the calling application, the calling application can retrieve specific information about the current optimization from the routine `CPXgetcallbackinfo()`. Optionally, the calling application can then tell ILOG CPLEX to discontinue optimization.

To implement and use a callback in your application, you must first write the callback function and then tell ILOG CPLEX about it. For more information about the ILOG CPLEX Callable Library routines for callbacks, see the *ILOG CPLEX Reference Manual*.

Setting Callbacks

In the Callable Library, diagnostic callbacks are grouped into two groups: LP callbacks and MIP callbacks. For each group, one callback function can be set, by calling functions `CPXsetlpcallbackfunc()` and `CPXsetmipcallbackfunc()`, respectively. You can distinguish between the actual callbacks by querying the parameter `wherefrom` which is passed to the callback function as parameter by ILOG CPLEX.

Callbacks for LPs and for MIPs

ILOG CPLEX will evaluate two user-defined callback functions, one during the solution of LP and QP problems and one during the solution of MIP problems. ILOG CPLEX calls the LP callback once per iteration during the solution of an LP or QP problem and periodically during the presolve. ILOG CPLEX calls the MIP callback periodically during the probing phase of MIP preprocessing, periodically during cut generation, and once before each subproblem is solved in the branch and cut process.

Every user-defined callback must have these arguments:

- `env`, a pointer to the ILOG CPLEX environment;
- `cbdata`, a pointer to ILOG CPLEX internal data structures needed by `CPXgetcallbackinfo()`;
- `wherefrom`, indicates which optimizer is calling the callback;
- `cbhandle`, a pointer supplied when your application calls `CPXsetlpcallbackfunc()` or `CPXsetmipcallbackfunc()` (so that the callback has access to private user data).

The arguments `wherefrom` and `cbdata` should be used only in calls to `CPXgetcallbackinfo()`.

Return Values for Callbacks

A user-written callback should return a nonzero value if the user wishes to stop the optimization and a value of zero otherwise.

For LP or QP problems, if the callback returns a nonzero value, the solution process will terminate. If the process was not terminated during the presolve process, the status returned by the function `IloCplex::getStatus` or the routines `CPXsolution()` or `CPXgetstat()` will be `CPX_STAT_ABORT_USER` (value `13`).

For both LP and MIP problems, if the LP callback returns a nonzero value during presolve preprocessing, the optimizer will return the value `CPXERR_PRESLV_ABORT`, and no solution information will be available.

For MIP problems, if the callback returns a nonzero value, the solution process will terminate and the status returned by `IloCplex::getStatus()` or `CPXgetstat()` will be one of the values in Table 9.5.

Table 9.5 *Status of Nonzero Callbacks for MIPs*

Value	Symbolic constant	Meaning
113	CPXMIP_ABORT_FEAS	current solution integer feasible
114	CPXMIP_ABORT_INFEAS	no integer feasible solution found

Interaction Between Callbacks and ILOG CPLEX Parallel Optimizers

When you use callback routines, and invoke the parallel version of ILOG CPLEX optimizers, you need to be aware that the ILOG CPLEX environment passed to the callback routine corresponds to an individual ILOG CPLEX thread rather than to the original environment created. ILOG CPLEX frees this environment when finished with the thread. This does not affect most uses of the callback function. However, keep in mind that ILOG CPLEX associates problem objects, parameter settings, and message channels with the environment that specifies them. ILOG CPLEX therefore frees these items when it removes that environment; if the callback uses routines like `CPXcreateprob`, `CPXcloneprob`, or `CPXgetchannels`, those objects remain allocated only as long as the associated environment does. Similarly, setting parameters with routines like `CPXsetintparam` affects settings only within the thread. So, applications that access ILOG CPLEX objects in the callback should use the original environment you created if they need to access these objects outside the scope of the callback function.

Example: Using Callbacks

This example shows you how to use callbacks effectively with routines from the ILOG CPLEX Callable Library. It is based on `lpex1.c`, a program described in the ILOG CPLEX *Getting Started* manual. This example about callbacks differs from that simpler one in several ways:

- To make the output more interesting, this example optimizes a slightly different linear program.
- The ILOG CPLEX screen indicator (that is, the parameter `CPX_PARAM_SCRIND`) is not turned on. Only the callback function produces output. Consequently, this program calls `CPXgeterrorstring()` to determine any error messages and then prints them. After the `TERMINATE:` label, the program uses separate status variables so that if an error

occurred earlier, its error status will not be lost or destroyed by freeing the problem object and closing the ILOG CPLEX environment. Table 9.6 summarizes those status variables.

Table 9.6 *Status Variables in* `lpex4.c`

Variable	Represents status returned by this routine
`frstatus`	`CPXfreeprob()`
`clstatus`	`CPXcloseCPLEX()`

- The function `mycallback()` at the end of the program is called by the optimizer. This function tests whether the primal simplex optimizer has been called. If so, then a call to `CPXgetcallbackinfo()` gets the following information:
  - iteration count;
  - feasibility indicator;
  - sum of infeasibilities (if infeasible);
  - objective value (if feasible).

  The function then prints these values to indicate progress.
- Before the program calls `CPXlpopt()`, the default optimizer from the ILOG CPLEX Callable Library, it sets the callback function by calling `CPXsetlpcallbackfunc()`. It unsets the callback immediately after optimization.

This callback function offers a model for graphic user interfaces that display optimization progress as well as those GUIs that allow a user to interrupt and stop optimization. If you want to provide your end-user a facility like that to interrupt and stop optimization, then you should make `mycallback()` return a nonzero value to indicate the end-user interrupt.

Complete Program: lpex4.c

The complete program, `lpex4.c`, appears here and online in the standard distribution.

```
/*------------------------------------------------------------------------*/
/*  File: examples/src/lpex4.c                                            */
/*  Version 8.1                                                           */
/*------------------------------------------------------------------------*/
/*  Copyright (C) 1997-2002 by ILOG.                                      */
/*  All Rights Reserved.                                                  */
/*  Permission is expressly granted to use this example in the            */
/*  course of developing applications that use ILOG products.             */
/*------------------------------------------------------------------------*/

/* lpex4.c - Illustrating the CPLEX callback functionality.

   This is a modification of lpex1.c, where we use a callback
   function to print the iteration info, rather than have CPLEX
```

```
   do it. */

/* Bring in the CPLEX function declarations and the C library
   header file stdio.h with the following single include. */

#include <ilcplex/cplex.h>

/* Bring in the declarations for the string functions */

#include <string.h>

/* Include declaration for function at end of program */

static int
   populatebycolumn  (CPXENVptr env, CPXLPptr lp);

static int CPXPUBLIC
   mycallback (CPXCENVptr env, void *cbdata, int wherefrom,
               void *cbhandle);



/* The problem we are optimizing will have 2 rows, 3 columns
   and 6 nonzeros.  */

#define NUMROWS    2
#define NUMCOLS    3
#define NUMNZ      6

int
main (void)
{
   char     probname[16];  /* Problem name is max 16 characters */

   /* Declare and allocate space for the variables and arrays where we
      will store the optimization results including the status, objective
      value, variable values, dual values, row slacks and variable
      reduced costs. */

   int      solstat;
   double   objval;
   double   x[NUMCOLS];
   double   pi[NUMROWS];
   double   slack[NUMROWS];
   double   dj[NUMCOLS];


   CPXENVptr     env = NULL;
   CPXLPptr      lp = NULL;
   int           status;
   int           i, j;
   int           cur_numrows, cur_numcols;
```

```
/* Initialize the CPLEX environment */

env = CPXopenCPLEX (&status);

/* If an error occurs, the status value indicates the reason for
   failure.  The error message will be printed at the end of the
   program. */

if ( env == NULL ) {
   fprintf (stderr, "Could not open CPLEX environment.\n");
   goto TERMINATE;
}

/* Turn *off* output to the screen since we'll be producing it
   via the callback function.  This also means we won't see any
   CPLEX generated errors, but we'll handle that at the end of
   the program. */

status = CPXsetintparam (env, CPX_PARAM_SCRIND, CPX_OFF);
if ( status ) {
   fprintf (stderr,
            "Failure to turn off screen indicator, error %d.\n", status);
   goto TERMINATE;
}

/* Create the problem. */

strcpy (probname, "example");
lp = CPXcreateprob (env, &status, probname);

/* A returned pointer of NULL may mean that not enough memory
   was available or there was some other problem.  In the case of
   failure, an error message will have been written to the error
   channel from inside CPLEX.  In this example, we wouldn't see
   an error message from CPXcreateprob since we turned off the
   CPX_PARAM_SCRIND parameter above.  The only way to see this message
   would be to use the CPLEX message handler, but that clutters up
   the simplicity of this example, which has a point of illustrating
   the CPLEX callback functionality.   */

if ( lp == NULL ) {
   fprintf (stderr, "Failed to create LP.\n");
   goto TERMINATE;
}

/* Now populate the problem with the data. */

status = populatebycolumn (env, lp);

if ( status ) {
   fprintf (stderr, "Failed to populate problem data.\n");
   goto TERMINATE;
```

```
}

status = CPXsetlpcallbackfunc (env, mycallback, NULL);
if ( status ) {
   fprintf (stderr, "Failed to set callback function.\n");
   goto TERMINATE;
}

/* Optimize the problem and obtain solution. */

status = CPXsetintparam (env, CPX_PARAM_LPMETHOD, CPX_ALG_PRIMAL);
if ( status ) {
   fprintf (stderr,
            "Failed to set the optimization method, error %d.\n", status);
   goto TERMINATE;
}


status = CPXlpopt (env, lp);
if ( status ) {
   fprintf (stderr, "Failed to optimize LP.\n");
   goto TERMINATE;
}

/* Turn off the callback function.  This isn't strictly necessary,
   but is good practice.  Note that the cast in front of NULL
   is only necessary for some compilers.   */

status = CPXsetlpcallbackfunc (env,
           (int (CPXPUBLIC *)(CPXCENVptr, void *, int, void *)) NULL, NULL);

if ( status ) {
   fprintf (stderr, "Failed to turn off callback function.\n");
   goto TERMINATE;
}

status = CPXsolution (env, lp, &solstat, &objval, x, pi, slack, dj);
if ( status ) {
   fprintf (stderr, "Failed to obtain solution.\n");
   goto TERMINATE;
}


/* Write the output to the screen. */

printf ("\nSolution status = %d\n", solstat);
printf ("Solution value  = %f\n\n", objval);

/* The size of the problem should be obtained by asking CPLEX what
   the actual size is, rather than using sizes from when the problem
   was built.  cur_numrows and cur_numcols store the current number
   of rows and columns, respectively.  */
```

```
   cur_numrows = CPXgetnumrows (env, lp);
   cur_numcols = CPXgetnumcols (env, lp);
   for (i = 0; i < cur_numrows; i++) {
      printf ("Row %d:  Slack = %10f  Pi = %10f\n", i, slack[i], pi[i]);
   }

   for (j = 0; j < cur_numcols; j++) {
      printf ("Column %d:  Value = %10f  Reduced cost = %10f\n",
              j, x[j], dj[j]);
   }

   /* Finally, write a copy of the problem to a file. */

   status = CPXwriteprob (env, lp, "lpex4.lp", NULL);
   if ( status ) {
      fprintf (stderr, "Failed to write LP to disk.\n");
      goto TERMINATE;
   }


TERMINATE:

   /* Free up the problem as allocated by CPXcreateprob, if necessary */

   if ( lp != NULL ) {
      int  frstatus;
      frstatus = CPXfreeprob (env, &lp);
      if ( frstatus ) {
         fprintf (stderr, "CPXfreeprob failed, error code %d.\n", frstatus);
         if (( !status ) && frstatus )  status = frstatus;
      }
   }

   /* Free up the CPLEX environment, if necessary */

   if ( env != NULL ) {
      int  clstatus;
      clstatus = CPXcloseCPLEX (&env);

      if ( clstatus ) {
         fprintf (stderr, "CPXcloseCPLEX failed, error code %d.\n", clstatus);
         if (( !status ) && clstatus )  status = clstatus;
      }
   }

   if ( status ) {
      char  errmsg[1024];

      /* Note that since we have turned off the CPLEX screen indicator,
         we'll need to print the error message ourselves. */

      CPXgeterrorstring (env, status, errmsg);
```

```
      fprintf (stderr, "%s", errmsg);
   }

   return (status);

}  /* END main */


/* This function builds by column the linear program:

      Maximize
       obj: x1 + 2 x2 + 3 x3
      Subject To
       c1: - x1 + x2 + x3 <= 20
       c2: x1 - 3 x2 + x3 <= 30
      Bounds
       0 <= x1 <= 40
      End
 */

static int
populatebycolumn (CPXENVptr env, CPXLPptr lp)
{
   int      status    = 0;
   double   obj[NUMCOLS];
   double   lb[NUMCOLS];
   double   ub[NUMCOLS];
   char     *colname[NUMCOLS];
   int      matbeg[NUMCOLS];
   int      matind[NUMNZ];
   double   matval[NUMNZ];
   double   rhs[NUMROWS];
   char     sense[NUMROWS];
   char     *rowname[NUMROWS];

   /* To build the problem by column, create the rows, and then
      add the columns. */

   CPXchgobjsen (env, lp, CPX_MAX);  /* Problem is maximization */

   /* Now create the new rows.  First, populate the arrays. */

   rowname[0] = "c1";
   sense[0]   = 'L';
   rhs[0]     = 20.0;

   rowname[1] = "c2";
   sense[1]   = 'L';
   rhs[1]     = 30.0;

   status = CPXnewrows (env, lp, NUMROWS, rhs, sense, NULL, rowname);
   if ( status )   goto TERMINATE;
```

```
   /* Now add the new columns.  First, populate the arrays. */

       obj[0] = 1.0;      obj[1] = 2.0;           obj[2] = 3.0;

    matbeg[0] = 0;     matbeg[1] = 2;          matbeg[2] = 4;

    matind[0] = 0;     matind[2] = 0;          matind[4] = 0;
    matval[0] = -1.0;  matval[2] = 1.0;        matval[4] = 1.0;

    matind[1] = 1;     matind[3] = 1;          matind[5] = 1;
    matval[1] = 1.0;   matval[3] = -3.0;       matval[5] = 1.0;

        lb[0] = 0.0;       lb[1] = 0.0;           lb[2]  = 0.0;
        ub[0] = 40.0;      ub[1] = CPX_INFBOUND;  ub[2]  = CPX_INFBOUND;

   colname[0] = "x1"; colname[1] = "x2";      colname[2] = "x3";

   status = CPXaddcols (env, lp, NUMCOLS, NUMNZ, obj, matbeg, matind,
                        matval, lb, ub, colname);
   if ( status )  goto TERMINATE;

TERMINATE:

   return (status);

}  /* END populatebycolumn */


/* The callback function will print out the Phase of the simplex method,
   the sum of infeasibilities if in Phase 1, or the objective if in Phase 2.
   If any of our requests fails, we'll return an indication to abort.
 */

static int CPXPUBLIC
mycallback (CPXCENVptr env, void *cbdata, int wherefrom, void *cbhandle)
{
   int    status = 0;
   int    phase  = -1;
   double suminf_or_objective;
   int    itcnt = -1;

   if ( wherefrom == CPX_CALLBACK_PRIMAL ) {
      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_ITCOUNT, &itcnt);
      if ( status )  goto TERMINATE;

      status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                   CPX_CALLBACK_INFO_PRIMAL_FEAS, &phase);
      if ( status )  goto TERMINATE;

      if ( phase == 0 ) {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
```

```
                                        CPX_CALLBACK_INFO_PRIMAL_INFMEAS,
                                        &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf ("Iteration %d: Infeasibility measure = %f\n",
                  itcnt, suminf_or_objective);
      }
      else {
         status = CPXgetcallbackinfo (env, cbdata, wherefrom,
                                        CPX_CALLBACK_INFO_PRIMAL_OBJ,
                                        &suminf_or_objective);
         if ( status )  goto TERMINATE;

         printf ("Iteration %d: Objective = %f\n",
                  itcnt, suminf_or_objective);
      }

   }

TERMINATE:

   return (status);

} /* END mycallback */
```

Control Callbacks for IloCplex

Control callbacks allow you to control the branch & cut search during the optimization of MIP problems. The following control callbacks are available for `IloCplex`:

- The node callback allows you to query and optionally overwrite the next node ILOG CPLEX will process during a branch & cut search.
- The solve callback allows you to specify and configure the optimizer option to be used for solving the LP at each individual node.
- The cut callback allows you to add problem-specific cuts at each node.
- The heuristic callback allows you to implement a heuristic that tries to generate a new incumbent from the solution of the LP relaxation at each node.
- The branch callback allows you to query and optionally overwrite the way ILOG CPLEX will branch at each node.
- The incumbent callback allows you to check and optionally reject incumbents found by ILOG CPLEX during the search.

These callbacks are implemented as an extension of the diagnostic callback class hierarchy. This extension is shown below along with the macro names for each of the control callbacks

(see *Diagnostic Callbacks* on page 341 for a discussion of how macros and callback implementation classes relate).

```
IloCplex::MIPCallbackI                          ILOMIPCALLBACKn
   |
   +--- IloCplex::NodeCallbackI                 ILONODECALLBACKn
   |
   +--- IloCplex::IncumbentCallbackI            ILOINCUMBENTCALLBACKn
   |
   +--- IloCplex::ControlCallbackI              ILOCONTROLCALLBACKn
           |
           +--- IloCplex::BranchCallbackI       ILOBRANCHCALLBACKn
           |
           +--- IloCplex::CutCallbackI          ILOCUTCALLBACKn
           |
           +--- IloCplex::HeuristicCallbackI    ILOHEURISTICCALLBACKn
           |
           +--- IloCplex::SolveCallbackI        ILOSOLVECALLBACKn
```

Again, the callback class hierarchy for Java is exactly the same, but the class names differ, in that there is no 'I' at the end. For example the corresponding Java implementation class for `IloCplex::BranchCallbackI` is `IloCplex.BranchCallback`.

Similar to class `IloCplex::CallbackI` (`IloCplex.Callback`), class `IloCplex::ControlCallbackI` (`IloCplex.ControlCallback`) is not provided for deriving user callback classes, but for defining the common interface for its derived classes. This interface provides methods for querying information about the current node, such as current bounds or solution information for the current node. See class `IloCplex::ControlCallbackI` (`IloCplex.ControlCallback`) in the *ILOG CPLEX Reference Manual* for more information.

Example: Controlling Cuts

This example shows how to use the cut callback in the context of solving the *noswot* model. This is a relatively small model from the MIPLIB 3.0 test-set, consisting only of 128 variables. This model is very hard to solve by itself, in fact until the release of ILOG CPLEX 6.5 it appeared to be unsolvable even after days of computation.

While it is now solvable directly, the computation time is in the order of several hours on state-of-the-art computers. However, cuts can be derived, the addition of which make the problem solvable in a matter of minutes or seconds. These cuts are:

```
x21 - x22 <= 0
x22 - x23 <= 0
x23 - x24 <= 0
2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51 <= 20.25
2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52 <= 20.25
2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53 <= 20.25
```

```
  2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
  0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54 <= 20.25
  2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
  0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55 <= 16.25
```

These cuts have been derived after interpreting the model as a resource allocation model on five machines with scheduling, horizon constraints and transaction times. The first three cuts break symmetries among the machines, while the others capture minimum bounds on transaction costs. See *MIP: Theory and Practice — Closing the Gap*[1] for more information about how these cuts have been found.

Of course the best way to solve the *noswot* model with these cuts is to simply add the cuts to the model before calling the optimizer. However, for demonstration purposes, we will add the cuts, using a cut callback, only when they are violated at a node. This cut callback takes a list of cuts as parameter and adds individual cuts whenever they are violated with the current LP solution. Notice, that adding cuts does not change the extracted model, but affects only the internal problem representation of the ILOG CPLEX object.

Let us first consider the C++ version of the callback. In C++, the callback is implemented with the code:

```
ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
  IloInt n = lhs.getSize();
  for (IloInt i = 0; i < n; ++i) {
    IloNum xrhs = rhs[i];
    if ( xrhs < IloInfinity && getValue(lhs[i]) > xrhs + eps ) {
      IloRange cut;
      try {
        cut = (lhs[i] <= xrhs);
        add(cut).end();
        rhs[i] = IloInfinity;
      }
      catch (...) {
        cut.end();
        throw;
      }
    }
  }
}
```

This defines the class `CtCallbackI` as a derived class of `IloCplex::CutCallbackI` and provides the implementation for its virtual methods `main()` and `duplicateCallback()`. It also implements a function `CtCallback` that creates an instance of `CtCallbackI` and returns an `IloCplex::Callback` handle for it.

As indicated by the `3` in the macro name, the constructor of `CtCallbackI` takes three parameters, called `lhs`, `rhs`, and `eps`. The constructor stores them as private members to have direct access to them in the callback function, implemented as method `main`. Notice

1. http://www.ilog.com/products/optimization/tech/researchpapers.cfm#MIPTheory

the comma (,) between the type and the argument object in the macro invocation. Here is how the macro expands:

```
class CtCallbackI : public IloCplex::CutCallbackI {
  IloExprArray lhs;
  IloNumArray  rhs;
  IloNum       eps;
public:
  IloCplex::CallbackI* duplicateCallback() const {
    return (new (getEnv()) CtCallbackI(*this));
  }
  CtCallbackI(IloExprArray xlhs, IloNumArray xrhs, IloNum xeps)
    : lhs(xlhs), rhs(xrhs), eps(xeps)
  {}
  void main();
};

IloCplex::Callback CtCallback(IloEnv env,
                              IloExprArray lhs,
                              IloNumArray rhs,
                              IloNum eps) {
  return (IloCplex::Callback(new (env) CtCallbackI(lhs, rhs, eps)));
}

void CtCallbackI::main() {
  ...
}
```

where the actual implementation code has been substituted with "...". Similar macros are provided for other numbers of parameters ranging from 0 through 7 for all callback classes.

The first parameter `lhs` is an array of expressions, and the parameter `rhs` is an array of values. These parameters are the left-hand side and right-hand side values of cuts of the form `lhs` $\leq$ `rhs` to be tested for violation and potentially added. The third parameter `eps` gives a tolerance by which a cut must at least be violated in order to be added to the problem being solved.

The implementation of this example cut callback looks for cuts that are violated by the current LP solution of the node where the callback is invoked. We loop over the potential cuts, checking each for violation by querying the value of the `lhs` expression with respect to the current solution. This is done by calling `getValue()` with this expression as a parameter. This is tested for violation of more than the tolerance parameter `eps` with the corresponding right-hand side value.

> ***Tip:*** A numerical tolerance is always a wise thing to consider when dealing with any non-trivial model, to avoid certain logical inconsistencies that could otherwise occur due to numerical roundoff. Here we use the standard ILOG CPLEX simplex feasibility tolerance for this purpose, to insure some consistency with the way ILOG CPLEX is treating the rest of the model.

If a violation is detected, the callback creates an `IloRange` object to represent the cut: `lhs[i]` ≤ `rhs[i]`. It is added to the LP by calling method `add()`. Adding a cut to ILOG CPLEX, unlike extracting a model, only copies the cut into the ILOG CPLEX data structures, without maintaining a notification link between the two. Thus, after a cut has been added, it can be deleted by calling its method `end()`. In fact, it should be deleted, as otherwise the memory used for the cut could not be reclaimed. For convenience, method `add()` returns the cut that has been added, and thus we can call `end()` directly on the returned `IloRange` object.

It is important that all resources that have been allocated during a callback are freed again before leaving the callback--even in the case of an exception. Here exceptions could be thrown when creating the cut itself or when trying to add it, for example, due to memory exhaustion. Thus, we enclose these operations in a try block and catch all exceptions that may occur. In the case of an exception, we delete the cut by calling `cut.end()` and re-throw whatever exception was caught. Re-throwing the exception can be omitted if you want to continue the optimization without the cut.

After the cut has been added, we set the `rhs` value to `IloInfinity` to avoid checking this cut for violation at the next invocation of the callback. Note that we did not simply remove the *i*th element of arrays `rhs` and `lhs`, because this is not supported if the cut callback is invoked from a parallel optimizer. However, changing array elements is allowed.

Also, for the potential use of the callback in parallel, the variable `xrhs` ensures that we are using the same value when checking for violation of the cut as when adding the cut. Otherwise, another thread may have set the `rhs` value to `IloInfinity` just between the two actions, and a useless cut would be added. ILOG CPLEX would actually handle this correctly, as it handles adding the same cut from different threads.

Function `makeCuts()` generates the arrays `rhs` and `lhs` to be passed to the cut callback. It first declares the array of variables to be used for defining the cuts. Since the environment is not passed to the constructor of that array, an array of 0-variable handles is created. In the following loop, these variable handles are initialized to the correct variables in the `noswot` model which are passed to this function as parameter `vars`. The identification of the variables is done by querying variables names. Once all the variables have been assigned, they are used to create the `lhs` expressions and `rhs` values of the cuts.

The cut callback is created and passed to ILOG CPLEX in the line:

```
cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
```

The function `CtCallback` constructs an instance of our callback class `CtCallbackI` and returns an `IloCplex::Callback` handle object for it. This is directly passed to function `cplex.use`.

The Java implementation of the callback is quite similar:

```
public static class Callback extends IloCplex.CutCallback {
   double     eps = 1.0e-6;
   IloRange[] cut;
   Callback(IloRange[] cuts) { cut = cuts; }

   public void main() throws IloException {
      int num = cut.length;
      for (int i = 0; i < num; ++i) {
         if ( cut[i] != null ) {
            double val = getValue(cut[i].getExpr());
            if ( cut[i].getLB() > val+eps || val-eps > cut[i].getUB() ) {
               add(cut[i]);
               cut[i] = null;
            }
         }
      }
   }
}
```

Instead of receiving expressions and right-hand side values, we chose to directly pass an array of `IloRange` constraints to the callback which are stored in member `cut`. The `main()` method loops over all cuts and evaluates the constraint expressions at the current solution by calling `getValue(cut[i].getExpr())`. If this exceeds the constraint bounds by more than `eps`, the cut is added during the search by calling `add(cut[i])` and `cut[i]` is set to `null` to avoid unneccessarily evaluating it again.

If for the C++ version, the array of cuts passed to the callback is initialized in a separate function `makeCuts()`. The callback is then created and used to with the `noswot` cuts by calling.

```
cplex.use(new Callback(makeCuts(cplex, lp)));
```

We should point out that `IloCplex` provides an easier way to manage such cuts in a case like this, where all cuts can be easily enumerated before starting the optimization. Calling the methods `cplex.addCut()` and `cplex.addCuts()` allows you to copy the cuts to `IloCplex` before the optimization. Thus, instead of creating and using the callback, we could have written:

```
cplex.addCuts(makeCuts(var));
```

as shown in example `iloadmipex4.cpp` in the distribution. During branch & cut, ILOG CPLEX will consider adding individual cuts to its representation of the model only if they are violated by a node LP solution in about the same way this example handles them. Whether this or adding the cuts directly to the model gives better performance when solving the model depends on the individual problem.

Complete Program: iloadmipex5.cpp

The complete program, `iloadmipex5.cpp`, appears here and online in the standard distribution.The Java version is found in file `AdMIPex5.java` at the same location.

```
// ------------------------------------------------------------- -*- C++ -*-
// File: examples/src/iloadmipex5.cpp
// Version 8.1
// --------------------------------------------------------------------------
//  Copyright (C) 1999-2002 by ILOG.
//  All Rights Reserved.
//  Permission is expressly granted to use this example in the
//  course of developing applications that use ILOG products.
// --------------------------------------------------------------------------
//
// iloadmipex5.cpp -- Solving noswot by adding cuts in a cut callback
//

#include <ilcplex/ilocplex.h>
ILOSTLBEGIN

ILOCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
   IloInt n = lhs.getSize();
   for (IloInt i = 0; i < n; ++i) {
      IloNum xrhs = rhs[i];
      if ( xrhs < IloInfinity  &&  getValue(lhs[i]) > xrhs + eps ) {
         IloRange cut;
         try {
            cut = (lhs[i] <= xrhs);
            add(cut).end();
            rhs[i] = IloInfinity;
         }
         catch (...) {
            cut.end();
            throw;
         }
      }
   }
}

void
makeCuts(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs) {
   IloNumVar x11, x12, x13, x14, x15;
   IloNumVar x21, x22, x23, x24, x25;
   IloNumVar x31, x32, x33, x34, x35;
   IloNumVar x41, x42, x43, x44, x45;
   IloNumVar x51, x52, x53, x54, x55;
   IloNumVar w11, w12, w13, w14, w15;
   IloNumVar w21, w22, w23, w24, w25;
   IloNumVar w31, w32, w33, w34, w35;
   IloNumVar w41, w42, w43, w44, w45;
   IloNumVar w51, w52, w53, w54, w55;
   IloInt num = vars.getSize();
```

```
for (IloInt i = 0; i < num; ++i) {
   if      ( strcmp(vars[i].getName(), "X11") == 0 ) x11 = vars[i];
   else if ( strcmp(vars[i].getName(), "X12") == 0 ) x12 = vars[i];
   else if ( strcmp(vars[i].getName(), "X13") == 0 ) x13 = vars[i];
   else if ( strcmp(vars[i].getName(), "X14") == 0 ) x14 = vars[i];
   else if ( strcmp(vars[i].getName(), "X15") == 0 ) x15 = vars[i];
   else if ( strcmp(vars[i].getName(), "X21") == 0 ) x21 = vars[i];
   else if ( strcmp(vars[i].getName(), "X22") == 0 ) x22 = vars[i];
   else if ( strcmp(vars[i].getName(), "X23") == 0 ) x23 = vars[i];
   else if ( strcmp(vars[i].getName(), "X24") == 0 ) x24 = vars[i];
   else if ( strcmp(vars[i].getName(), "X25") == 0 ) x25 = vars[i];
   else if ( strcmp(vars[i].getName(), "X31") == 0 ) x31 = vars[i];
   else if ( strcmp(vars[i].getName(), "X32") == 0 ) x32 = vars[i];
   else if ( strcmp(vars[i].getName(), "X33") == 0 ) x33 = vars[i];
   else if ( strcmp(vars[i].getName(), "X34") == 0 ) x34 = vars[i];
   else if ( strcmp(vars[i].getName(), "X35") == 0 ) x35 = vars[i];
   else if ( strcmp(vars[i].getName(), "X41") == 0 ) x41 = vars[i];
   else if ( strcmp(vars[i].getName(), "X42") == 0 ) x42 = vars[i];
   else if ( strcmp(vars[i].getName(), "X43") == 0 ) x43 = vars[i];
   else if ( strcmp(vars[i].getName(), "X44") == 0 ) x44 = vars[i];
   else if ( strcmp(vars[i].getName(), "X45") == 0 ) x45 = vars[i];
   else if ( strcmp(vars[i].getName(), "X51") == 0 ) x51 = vars[i];
   else if ( strcmp(vars[i].getName(), "X52") == 0 ) x52 = vars[i];
   else if ( strcmp(vars[i].getName(), "X53") == 0 ) x53 = vars[i];
   else if ( strcmp(vars[i].getName(), "X54") == 0 ) x54 = vars[i];
   else if ( strcmp(vars[i].getName(), "X55") == 0 ) x55 = vars[i];
   else if ( strcmp(vars[i].getName(), "W11") == 0 ) w11 = vars[i];
   else if ( strcmp(vars[i].getName(), "W12") == 0 ) w12 = vars[i];
   else if ( strcmp(vars[i].getName(), "W13") == 0 ) w13 = vars[i];
   else if ( strcmp(vars[i].getName(), "W14") == 0 ) w14 = vars[i];
   else if ( strcmp(vars[i].getName(), "W15") == 0 ) w15 = vars[i];
   else if ( strcmp(vars[i].getName(), "W21") == 0 ) w21 = vars[i];
   else if ( strcmp(vars[i].getName(), "W22") == 0 ) w22 = vars[i];
   else if ( strcmp(vars[i].getName(), "W23") == 0 ) w23 = vars[i];
   else if ( strcmp(vars[i].getName(), "W24") == 0 ) w24 = vars[i];
   else if ( strcmp(vars[i].getName(), "W25") == 0 ) w25 = vars[i];
   else if ( strcmp(vars[i].getName(), "W31") == 0 ) w31 = vars[i];
   else if ( strcmp(vars[i].getName(), "W32") == 0 ) w32 = vars[i];
   else if ( strcmp(vars[i].getName(), "W33") == 0 ) w33 = vars[i];
   else if ( strcmp(vars[i].getName(), "W34") == 0 ) w34 = vars[i];
   else if ( strcmp(vars[i].getName(), "W35") == 0 ) w35 = vars[i];
   else if ( strcmp(vars[i].getName(), "W41") == 0 ) w41 = vars[i];
   else if ( strcmp(vars[i].getName(), "W42") == 0 ) w42 = vars[i];
   else if ( strcmp(vars[i].getName(), "W43") == 0 ) w43 = vars[i];
   else if ( strcmp(vars[i].getName(), "W44") == 0 ) w44 = vars[i];
   else if ( strcmp(vars[i].getName(), "W45") == 0 ) w45 = vars[i];
   else if ( strcmp(vars[i].getName(), "W51") == 0 ) w51 = vars[i];
   else if ( strcmp(vars[i].getName(), "W52") == 0 ) w52 = vars[i];
   else if ( strcmp(vars[i].getName(), "W53") == 0 ) w53 = vars[i];
   else if ( strcmp(vars[i].getName(), "W54") == 0 ) w54 = vars[i];
   else if ( strcmp(vars[i].getName(), "W55") == 0 ) w55 = vars[i];
}
```

```
   lhs.add(x21 - x22);
   rhs.add(0.0);

   lhs.add(x22 - x23);
   rhs.add(0.0);

   lhs.add(x23 - x24);
   rhs.add(0.0);

   lhs.add(2.08*x11 + 2.98*x21 + 3.47*x31 + 2.24*x41 + 2.08*x51 +
           0.25*w11 + 0.25*w21 + 0.25*w31 + 0.25*w41 + 0.25*w51);
   rhs.add(20.25);

   lhs.add(2.08*x12 + 2.98*x22 + 3.47*x32 + 2.24*x42 + 2.08*x52 +
           0.25*w12 + 0.25*w22 + 0.25*w32 + 0.25*w42 + 0.25*w52);
   rhs.add(20.25);

   lhs.add(2.08*x13 + 2.98*x23 + 3.47*x33 + 2.24*x43 + 2.08*x53 +
           0.25*w13 + 0.25*w23 + 0.25*w33 + 0.25*w43 + 0.25*w53);
   rhs.add(20.25);

   lhs.add(2.08*x14 + 2.98*x24 + 3.47*x34 + 2.24*x44 + 2.08*x54 +
           0.25*w14 + 0.25*w24 + 0.25*w34 + 0.25*w44 + 0.25*w54);
   rhs.add(20.25);

   lhs.add(2.08*x15 + 2.98*x25 + 3.47*x35 + 2.24*x45 + 2.08*x55 +
           0.25*w15 + 0.25*w25 + 0.25*w35 + 0.25*w45 + 0.25*w55);
   rhs.add(16.25);
}


int
main(int argc, char** argv)
{
   IloEnv env;
   try {
      IloModel m;
      IloCplex cplex(env);

      IloObjective   obj;
      IloNumVarArray var(env);
      IloRangeArray  con(env);

      env.out() << "reading ../../../examples/data/noswot.mps" << endl;
      cplex.importModel(m, "../../../examples/data/noswot.mps", obj, var, con);

      env.out() << "constructing cut callback ..." << endl;

      IloExprArray lhs(env);
      IloNumArray  rhs(env);
      makeCuts(var, lhs, rhs);
```

```
      cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));

      env.out() << "extracting model ..." << endl;
      cplex.extract(m);

      cplex.setParam(IloCplex::MIPInterval, 1000);
      env.out() << "solving model ...\n";
      cplex.solve();
      env.out() << "solution status is " << cplex.getStatus() << endl;
      env.out() << "solution value  is " << cplex.getObjValue() << endl;
   }
   catch (IloException& ex) {
      cerr << "Error: " << ex << endl;
   }
   env.end();
   return 0;
}
```

A Comparison of Goals and Callbacks

Goals and callbacks both provide an API within `IloCplex` to allow you to take control over the branch & cut search for solving MIP models. With one exception, the same functionality is available in both APIs. In fact, the goal API is built on top of callbacks. As a consequence, you cannot use callbacks and goals at the same time. To help you choose which API is more suited to your needs, this section examines commonalities and differences between both.

As pointed out previously, both APIs allow you to control the branch & cut search used by `IloCplex` to solve MIP models. The following points distinguish specific features of this control.

- Checking feasibility
  - With goals, you can discontinue the search at a node by returning a `Fail` goal. Alternatively, you can continue searching, even though an integer feasible solution has been found, by returning another nonempty goal.
  - With callbacks, you can use method `prune()` of the branch callback to discontinue the search, and an incumbent callback to accept or reject integer feasible solutions.
- Creating branches
  - With goals, you create branches by using by using `Or` goals with local cut goals as parameters.
  - With callbacks, you create branches by using a branch callback.
- Adding local or global cuts
  - With goals, you can add global and local cuts by using global and local cut goals, respectively.
  - With callbacks, you need to implement either a cut callback (for global and local cuts) or a branch callback for branching on local cuts
- Injecting solution candidates
  - With goals, you inject solutions by using a solution goal.
  - With callbacks, you need to implement a heuristic callback to inject solutions.
- Controlling the node selection strategy
  - With goals, you control node selection by applying node evaluators to your search goal.
  - With callbacks, you control node selection by using a node callback.

Thus, one of the main differences between goals and callbacks is that with goals, all functionality is available from the `execute()` method of the goal, whereas with callbacks, you must implement different callbacks to access different functionality.

As an example, suppose you want to extend a search to satisfy additional constraints that could not conveniently be added as linear constraints to the model.

With callbacks, you need to use an incumbent callback and a branch callback. The incumbent callback has to reject an otherwise integer feasible solution if it violates such an additional constraint. In this case, the branch callback has to follow up with an appropriate branch to enforce the constraint. Since the branch callback function allows branching only on individual variables, the determination of the appropriate branch may be quite difficult for constraints not modelled with linear expressions.

With goals, the feasibility test and the resulting branching can be implemented with a single goal.

The second big difference between goals and callbacks is that with goals you can easily specify different search strategies in different subtrees. To do this, simply provide different search goals as a parameter to the `Or` goal when creating the root nodes for the subtrees in question. To achieve a similar result with callbacks requires an implementation that is too complex for a presentation here.

The only functionality that is not supported via goals is that provided by the `solve` callback. Because of this, the `solve` callbacks can be used at the same time as goals. However, this callback is very rarely used.

In summary, goals can be advantageous if you want to take control over several steps of the branch & cut search simultaneously, or if you want to specify different search strategies in different subtrees. On the other hand, if you only need to control a single aspect of the search—for example, adding cuts—using the appropriate callback may involve a smaller API and thus be quicker and easier to understand and implement.

Using Parallel Optimizers

This section tells you how to use ILOG CPLEX parallel optimizers. It includes sections on:

- Threads
- Nondeterminism
- Clock Settings and Time Measurement
- Using Parallel Optimizers in the Interactive Optimizer
- Using Parallel Optimizers in the ILOG CPLEX Component Libraries
- Parallel MIP Optimizer
- Parallel Barrier Optimizer
- Parallel Simplex Optimizer

There are three specialized ILOG CPLEX optimizers—Parallel Simplex, Parallel MIP, and Parallel Barrier—implemented to run on hardware platforms with parallel processors. These parallel optimizers, like other ILOG CPLEX optimizers, are available in the Interactive Optimizer and in the Component Libraries, if you hold a ILOG CPLEX Parallel license.

Parallel barrier and MIP optimizers are included with all ILOG CPLEX libraries. The parallel dual simplex optimizer is only available in OpenMP® versions of the libraries (which will have `_omp` in the platform name). Additional compiler/linker flags may be needed when using the OpenMP versions of the libraries. Consult the `makefile` that is provided in the ILOG CPLEX distribution for your computer platform.

Threads

The ILOG CPLEX parallel optimizers are licensed for a specific maximum number of *threads* (that is, the number processors applied to a given problem). The number of threads that ILOG CPLEX actually uses during a parallel optimization is the *smaller* of:

- the number of threads made available by the operating system;
- the number of threads indicated by the *licensed* values of the thread-limit parameters. Table 9.7 summarizes the values of those thread-limit parameters.

Table 9.7 *Thread-Limit Parameters*

Interactive Commands	Concert Technology Enumeration Value	Callable Library Parameter
`set threads`	`Threads`	`CPX_PARAM_THREADS`
`set barrier limits threads`	`BarThreads`	`CPX_PARAM_BARTHREADS`

Table 9.7 *Thread-Limit Parameters*

Interactive Commands	Concert Technology Enumeration Value	Callable Library Parameter
`set simplex limits threads`	`SimThreads`	`CPX_PARAM_SIMTHREADS`
`set mip limits threads`	`MIPThreads`	`CPX_PARAM_MIPTHREADS`
`set mip limits strongthreads`	`StrongThreadLim`	`CPX_PARAM_STRONGTHREADLIM`

The global thread parameter `Threads` establishes a default thread count for all parallel optimizers. Thread limits for specific optimizers can be set to values that differ from the global default (for example, by setting `IloCplex::MIPThreads`).

The notion of the number of threads used when running a parallel CPLEX optimizer is entirely separate from the limit on licensed uses. A typical ILOG CPLEX license permits one licensed use, that is a single concurrent execution on one licensed computer. If the license also contains the parallel option with a thread limit of, say, four (on a machine with at least four processors), that one concurrent execution of ILOG CPLEX can employ any number of parallel threads to increase performance, up to that limit of 4. A license with the parallel option, that additionally has a limit larger than one on the number of licensed uses, can support that many simultaneous executions of ILOG CPLEX, each with the licensed maximum number of parallel threads. In such a case, the operating system will manage any contention for processors.

The number of parallel threads used by an ILOG CPLEX optimizer is usually controlled by ILOG CPLEX parameter settings. For the OpenMP version, the number can also be controlled by environment variable `OMP_NUM_THREADS`. Set this variable to establish an upper limit on the number of threads used by ILOG CPLEX. This is discussed in more detail in the sections that follow.

Example: Threads and Licensing

For example, let's assume you use ILOG CPLEX to optimize MIP models on an eight processor machine, and you have purchased an ILOG CPLEX license for four parallel threads. Then you can use the Interactive Optimizer command `set threads i`, substituting values `1` through `4` for `i`. You will not be able to set the thread count higher than `4` because you are licensed for a maximum of four threads. If you set environment variable `OMP_NUM_THREADS` to `2` in the OpenMP version, then you can set the thread count to no more than `2`. Any MIP optimization you carry out will be limited to two processors.

Threads and Performance Considerations

If you set the number of threads to a value greater than the number of available processors, performance will usually degrade. If you set the number of threads to a value less than the number of processors, the remaining processors will be available for other jobs on your platform. Simultaneously running multiple parallel jobs with a total number of threads

exceeding the number of processors may impair the performance of each individual process as its threads compete with one another.

If you are using the OpenMP libraries and set the `OMP_NUM_THREADS` environment variable to a greater value than the actual thread count used with ILOG CPLEX, your operating system may create idle threads that still consume system resources. If you know in advance how many threads you want to use, we recommend that you set the environment variable to that number before you start ILOG CPLEX.

The benefit of applying more threads to optimizing a specific problem varies depending on the optimizer you use and the characteristics of the problem. You should experiment to assess performance improvements and degradation when you apply more or fewer processors. For example, when you optimize the root relaxation using the dual simplex method, there may be little or no benefit in applying more than four processors to the task. In contrast, if you use 16 processors during the MIP phase of an optimization, you may improve solution speed by a factor of 20. In such a case, you should set the parameters `simplex limit threads` and `mip limit threads` to *different* values in order to use your computing resources efficiently.

Another key consideration in setting optimizer and global thread limits is your management of overall system load.

Nondeterminism

Among the ILOG CPLEX parallel optimizers, only parallel simplex follows a deterministic algorithm, producing the same number of iterations and the same solution path when you apply it to the same problem more than once. In contrast, the parallel barrier and parallel MIP optimizers are *nondeterministic*: repeated solutions of a model using exactly the same settings can produce different solution paths and, in the case of the parallel MIP optimizer, very different solution times and results.

The basic algorithm in the ILOG CPLEX Parallel MIP Optimizer is branch & cut. The primary source of parallelism in branch & cut is the solution of the LP or QP subproblems at the individual nodes of the search tree. These subproblems can be distributed over available processors to be carried out in parallel. The individual solution paths for these subproblems will, in fact, be deterministic, but the speed at which their solutions occur can vary slightly. These variations lead to nodes being taken from and replaced in the branch & cut tree in different order, and this reordering leads to nondeterminism about many other quantities that control the optimization. This nondeterminism is unavoidable in such a context, and its effects can result in some cases in very different solution paths.

Clock Settings and Time Measurement

The clock-type parameter determines how ILOG CPLEX measures computation time. CPU time, the default, is most appropriate when only one processor is used. It reports the amount of time the CPU spent performing computation on behalf of your application. For parallel

execution, CPU time is system dependent and generally will not reflect the desired metric. On some parallel systems, it may measure aggregate CPU time, that is, the sum of time used by all processors. On others, it may report the CPU time of only one process. In short, it may give you a misleading indication of parallel speed.

The alternative type, wall-clock time, is usually more appropriate for parallel computing because it measures the total physical time elapsed after an operation begins. When multiple processes are active, and when parallel optimizers are active, wall-clock time can be much different from CPU time.

You can choose the type of clock setting, in the:

- Interactive Optimizer, with the command `set clocktype` *`i`*.
- Concert Technology Library, use the `IloCplex` method `setParam(ClockType,` *`i`*`)`.
- Callable Library, use the routine `CPXsetintparam(env, CPX_PARAM_CLOCKTYPE,` *`i`*`)`.

Replace the *`i`* with the value `1` to specify CPU time or `2` to specify wall-clock time.

Using Parallel Optimizers in the Interactive Optimizer

1. If necessary for your platform, set any environment variable needed for parallel operation. See *Threads* on page 371 for details.
2. Start the parallel CPLEX Interactive Optimizer with the command `cplex` at the operating system prompt.
3. Set the thread-limit, as explained in *Threads* on page 371.
4. Enter and populate your problem object as usual.
5. Call the parallel optimizer with the appropriate command:

Parallel MIP Optimizer	`mipopt`
Parallel Barrier Optimizer	`baropt`
Parallel Simplex Optimizer	`primopt` or `tranopt`

Using Parallel Optimizers in the ILOG CPLEX Component Libraries

1. Link your application to the parallel implementation of the ILOG CPLEX Component Libraries.
2. Create your ILOG CPLEX environment and initialize a problem object in the usual way. See *Initialize the ILOG CPLEX Environment* on page 85 and *Instantiate the Problem Object* on page 86 for details.
3. Within your application, set the appropriate ILOG CPLEX parameter from Table 9.7 to

specify the number of threads.

4. Enter and populate your problem object in the usual way, as in *Put Data in the Problem Object* on page 86.

5. Call the parallel optimizer with the appropriate method or routine:

Table 9.8 *Parallel Optimizer Methods and Routines*

Optimizer	IloCplex Method	Callable Library
Parallel MIP Optimizer	`solve()`	`CPXmipopt()`
Parallel Barrier Optimizer	`setParam(RootAlg, Barrier)`	`CPXbaropt()` or `CPXhybbaropt()`
Parallel Simplex Optimizer	`setParam(RootAlg, Primal)` or `setParam(RootAlg, Dual)`	`CPXprimopt()` or `CPXdualopt()`

Parallel MIP Optimizer

The ILOG CPLEX Parallel MIP Optimizer delivers significant increases in speed on a wide variety of models, particularly on difficult ones that solve a large number of nodes in the branch & cut search tree.

There are several different opportunities for applying multiple processors to the solution of a MIP problem.

Increase the Global Thread Parameter

The most straightforward way to invoke parallelism is by setting the global thread parameter, `Threads`, to a value greater than `1` to indicate the desired degree of parallelism. If the other parameters remain set to their default values, the result is that nodes in the branch & cut tree will be processed in parallel; that is, each node is solved in its entirety on one of the available processors, independently of the work being performed on other processors. In typical cases, the number of nodes waiting for solution quickly becomes greater than the number of threads, creating enough work which can be done in parallel to make speed increases possible. ILOG CPLEX automatically manages the pool of nodes, so that each time a thread finishes one node, it is assigned to solving another.

Branch & Cut Parallel Processing

A contrasting and specialized approach to obtaining speed increases by parallel processing within the branch & cut tree is to:

1. choose strong branching (`VarSel` parameter setting `3`) as the variable selection strategy;

2. apply multiple processors to the strong branching variable selection computation by setting the strong branching thread limit, `StrongThreadLim`, to a value greater than `1`; and

3. leaving the global thread limit at `1` to inhibit processing the branching and solution of the nodes in parallel.

On models where strong branching is already a beneficial technique, this approach to parallelism can be especially attractive.

Root Relaxation Parallel Processing

In some models, the LP or QP root relaxation takes significant solution time before parallel branching begins. These models have potential for additional speed increases by running the root relaxation step in parallel. If the root problem is an LP and the `Threads` parameter is set to a value greater than `1`, the concurrent optimizer is invoked by default. This provides a form of parallelism that applies the available threads to multiple LP optimizers. If the root problem is a QP, the dual simplex optimizer alone is used.

You can try a different form of parallelism at the root by selecting a specific optimizer with the starting algorithm parameter (`RootAlg` in Concert, `CPX_PARAM_STARTALG` in the Callable Library, "`set mip strategy startalgorithm`" in the Interactive Optimizer). The parallel threads will all be applied to the selected starting algorithm, to the extent that the algorithm supports parallelism. Parallel Barrier in particular is a frequent choice for this purpose. Later sections contain information on the Parallel Barrier Optimizer and the Parallel Simplex Optimizer. Note that if you use the barrier optimizer at the root of an MIQP problem, you must also use it for the nodes. This restriction does not apply in the case of MILP.

Individual Optimizer Parallel Processing

Parallelism in either barrier or simplex is ordinarily controlled by the global thread count parameter, but this default behavior can be overridden by the individual optimizer thread limit parameters, `BarThreads` and `SimThreads` respectively. The degree of parallelism within the branch & cut tree likewise can be controlled by the MIP thread limit `MipThreads`, which overrides the global thread limit. This capability to set any or all of `MipThreads`, `BarThreads`, and `SimThreads`, permits great flexibility in the use of parallel resources during the solution of a MIP model.

For example, on a model where only a small number of nodes is active at any one time, the benefit of parallel solution of these nodes is limited. If the individual nodes are computationally challenging, then it may be possible to achieve speed increases by leaving the global thread limit at its default of `1` and setting the continuous optimizer thread limit (`SimThreads` or `BarThreads`) to a value greater than `1`. The global thread limit of `1` will inhibit the parallelism of the branching process, while the explicit thread limit of more than `1` will permit the optimization of each node in parallel.

Nested Parallel Processing

Nested parallelism represents a further way to exploit the flexibility of independent thread parameters. For example it might be determined from experience, on a given family of models, that only a modest degree of parallelism is beneficial at the nodes and additional processors do not help speed up the branching. In such a case, better speed increases might be obtained by combining a parallelization of the work that the LP or QP optimizer does at

each node. On an 8-processor computer, you might opt to solve a model by setting the `MipThreads` limit to `4` instead of its maximum of `8`, and the `BarThreads` limit to `2`, thus keeping all `8` processors busy as much of the time as possible, with the four MIP threads each invoking two threads for the simplex optimizer.

If you do decide to try a nested parallel approach, keep in mind the rule of thumb that it is usually better to keep a higher degree of parallelism of the nodes themselves (`MipThreads`) than of the LP or QP optimizer (`SimThreads` or `Barthreads`); this is in keeping with the general observation that MIP speed increases are on average closer to linear in the number of threads than the speed increases for the continuous optimizers.

Note: *Nested parallelism is not supported by most OpenMP compilers and runtime libraries, so it is not available with the OpenMP version of ILOG CPLEX. On the OpenMP version you may set* `MipThreads` *to a value greater than* `1`*, or either of* `SimThreads` *or* `BarThreads` *to a value greater than* `1`*, but not both, in a single call to the MIP optimizer.*

Memory Considerations and the Parallel MIP Optimizer

Before the parallel MIP optimizer invokes parallel processing, it makes separate, internal copies of the initial problem. The individual processors use these copies during computation, so each of them requires an amount of memory roughly equal to the size of the original model after it is presolved.

Output from the Parallel MIP Optimizer

The parallel MIP optimizer generates slightly different output in the node log (see *Progress Reports: Interpreting the Node Log* on page 246) from the serial MIP optimizer. The following paragraphs explain those differences.

If the MIP optimizer is running in parallel, it will display elapsed time for the MIP optimizer in wall-clock time, independent of the setting of the clock-type parameter (assuming MIP logging has not been turned off).

ILOG CPLEX prints a summary of timing statistics specific to the parallel MIP optimizer at the end of optimization. You can see typical timing statistics in the following sample run.

```
Problem 'fixnet6.mps.gz' read.
Read time =    0.08 sec.
CPLEX> o
Tried aggregator 1 time.
MIP Presolve modified 308 coefficients.
Aggregator did 1 substitutions.
Reduced MIP has 477 rows, 877 columns, and 1754 nonzeros.
Presolve time =    0.02 sec.
Clique table members: 2
MIP emphasis: balance optimality and feasibility
Root relaxation solution time =    0.03 sec.

        Nodes                                         Cuts/
   Node  Left     Objective  IInf  Best Integer     Best Node    ItCnt     Gap

      0     0     3192.0420    12                 3192.0420      308
*     0+    0                   0     4505.0000     3192.0420      308   29.14%
                  3263.9220    19     4505.0000     Cuts:  36      344   27.55%
                  3393.0917    17     4505.0000     Cuts:  24      404   24.68%
*     0+    0                   0     4419.0000     3393.0917      404   23.22%
                  3439.4758    19     4419.0000  Flowcuts:  7      446   22.17%
                  3472.6583    20     4419.0000  Flowcuts:  6      462   21.42%
                  3489.9293    19     4419.0000  Flowcuts:  5      488   21.02%
                  3498.4754    26     4419.0000  Flowcuts:  5      506   20.83%
                  3499.7388    26     4419.0000  Flowcuts:  5      521   20.80%
                  3500.8733    26     4419.0000  Flowcuts:  3      531   20.78%
                  3526.2044    19     4419.0000  Flowcuts:  4      547   20.20%
                  3526.3698    21     4419.0000  Flowcuts:  1      548   20.20%
*     0+    0                   0     4310.0000     3526.3698      548   18.18%
*     0+    0                   0     3985.0000     3526.3698      548   11.51%
*    33     2                   0     3983.0000     3791.7448     1130    4.80%

Sequential (before b&c):
  Real     time             =    0.55
Parallel b&c, 2 threads:
  Real     time             =    0.16
  Critical time (total)     =    0.00
  Spin     time (average)   =    0.02
                             -------
Total (sequential+parallel) =    0.71 sec.

Cover cuts applied:  1
Flow cuts applied:  45
Gomory fractional cuts applied: 6

Integer optimal solution:  Objective =    3.9830000000e+03
Solution time =    0.72 sec.  Iterations = 1187  Nodes = 41
```

The summary at the end of the sample tells us that 0.55 seconds were spent in the sequential phase, that is, all the combined steps (preprocessing, root relaxation solution, cutting planes, heuristic) that occur at the root before the first branch occurs. The parallel part of this sample run took 0.16 seconds of real time (that is, elapsed time for that phase).

Other parts of the sample report indicate that the processors spent an average of *0.02* seconds of real time spinning (that is, waiting for work while there were too few active

nodes available). The real critical time was a total of *0.00* seconds, time spent by individual processors in updating global, shared information. Since only one processor can access the critical region at any instant in time, the amount of time spent in this region really is crucial: any other processor that tries to access this region must wait, thus sitting idle, and this idle time is counted separately from the spin time.

There is another difference in the way logging occurs in the parallel MIP optimizer. When this optimizer is called, it makes a number of copies of the problem. These copies are known as *clones*. The parallel MIP optimizer creates as many clones as there are threads available to it. When the optimizer exits, these clones and the memory they used are discarded.

If a log file is active when the clones are created, then ILOG CPLEX creates a clone log file for each clone. The clone log files are named `cloneK.log`, where `K` is the index of the clone, ranging from 0 (zero) to the number of threads minus one. Since the clones are created at each call to the parallel MIP optimizer and discarded when it exits, the clone logs are opened at each call and closed at each exit. (The clone log files are not removed when the clones themselves are discarded.)

The clone logs contain information normally recorded in the ordinary log file (by default, `cplex.log`) but inconvenient to send through the normal log channel. The information likely to be of most interest to you are special messages, such as error messages, that result from calls to the LP optimizers called for the subproblems.

Parallel Barrier Optimizer

The ILOG CPLEX Parallel Barrier Optimizer achieves significant speedups over its serial counterpart (the ILOG CPLEX Barrier Optimizer described in *Solving LP Problems with the Barrier Optimizer* on page 156, and in Chapter 7, *Solving Quadratic Programming Problems*) on a wide variety of classes of problems. Consequently, the parallel barrier optimizer will be the best LP choice on a parallel computer more frequently than on a single-processor. For that reason, you should be careful *not* to apply performance data or experience based on *serial* optimizers when you are choosing which optimizer to use on a parallel platform.

If you decide to use the parallel barrier optimizer on the *subproblems* of a MIP, see also other special considerations about nested parallelism in *Nested Parallel Processing* on page 376.

Parallel Simplex Optimizer

In the ILOG CPLEX implementation of the Parallel Simplex Optimizer, the column-based work occurs in parallel. Consequently, a significant gain in speed may occur in the dual simplex optimization(`tranopt` / `IloCplex::Dual` / `CPXdualopt()`).

Occasionally, in the primal simplex optimizer if the primal gradient parameter is set to steepest-edge pricing and the aspect ratio (that is, the number of columns divided by the number of rows) is relatively large (for example, *10* or more), then good speedups may

occur with the parallel optimizer here, too. Larger problems with a somewhat smaller aspect ratio may also benefit from parallel simplex optimization. Since it is difficult to predict where the breakpoint will be, we encourage you to experiment.

If you decide to use the parallel simplex optimizer on the *subproblems* of a MIP, see other special considerations about nested parallelism in *Using Parallel Optimizers* on page 371.

Platform Considerations

The parallel simplex optimizer is only available in the OpenMP version of ILOG CPLEX, which is not available on all platforms.

CPLEX Runtime Development and Licensing Procedures

ILOG CPLEX uses the standard ILOG License Manager (ILM). The *ILOG License Manager* online documentation, which comes with your software distribution, describes ILM access keys (or *keys*, for short) in more detail. This section describes how to write applications that use ILM runtime access keys.

An ILOG CPLEX runtime license costs significantly less money than a regular development license. However, its usage is restricted to applications created by a particular developer or company. In order to distinguish runtime access keys from development keys (as well as runtime keys for applications created by other companies), you need to call an additional routine in your source code before initializing the CPLEX environment.

Types of ILM Runtime Licenses

ILM runtime licenses come in two forms: file-based and memory-based.

File-Based RTNODE, RTSTOKEN or TOKEN Keys.

These are a file-based access key that is tied to a particular computer or server. Refer to the *ILOG License Manager* online documentation, which can be found in your distribution source, for information regarding how to establish the file containing the key. You must communicate the location of this file to your application. In order to avoid potential conflicts with other runtime applications, we recommend that you locate the key in a directory specific to your application by using the `CPXputenv()` routine in C or C++, or `IloCplex.putenv()` in java. These are the most commonly used runtime licenses.

Memory-Based RUNTIME Keys.

These involve passing some information in the memory of your program to ILM. No files containing access keys are involved. Rather, you set the key in your program and pass it to ILM by calling the `CPXRegisterLicense()` routine in C or C++, or `IloCplex.registerLicense()` in java.

Routines Used for Licensing

All ILOG CPLEX applications either call the `CPXopenCPLEX()` routine to establish the CPLEX environment, or use the `IloCplex` constructor to initialize ILOG CPLEX for use with Concert Technology. Until either `CPXopenCPLEX()` is called or the `IloCplex` object exists, few ILOG CPLEX routines operate. In addition to allocating the environment, `CPXopenCPLEX()` performs license checks, as does the constructor for the `IloCplex` object. For development licenses, no additional licensing steps are required. For runtime licenses, your application first needs to provide some additional licensing information before the call to `CPXopenCPLEX()` or the use of an `IloCplex` constructor. For `RTNODE`, `RTSTOKEN` and `TOKEN` keys, this requires calling the `CPXputenv()` routine from the

Callable Library and C++ Concert, or the `IloCplex.putenv()` static method from Java, to specify the location of the key through the `ILOG_LICENSE_FILE` environment variable. For memory-based `RUNTIME` keys, this requires calling the `CPXRegisterLicense()` routine for Callable Library and C++ users, or the static method `IloCplex.registerLicense()` for Java users, to pass the `RUNTIME` key to ILM.

Descriptions of the `CPXputenv()` and `CPXRegisterLicense()` routines are in the *ILOG CPLEX Reference Manual*; descriptions of `IloCplex.putenv()` and `IloCplex.registerLicense()` are in the *ILOG CPLEX Java Reference Manual*.

Examples

Here are some code samples that illustrate the usage of the above runtime license routines. The first example illustrates usage of the `CPXputenv()` routine when opening the CPLEX environment.

> ***Notes:*** *This example assumes a Microsoft Windows file directory structure that requires an additional backslash when specifying the path of the file containing the key. It also assumes that the application uses an environment variable called* `MYAPP_HOME` *to identify the directory in which it was installed.*
>
> *The string argument to* `CPXputenv()` *must remain active throughout the time ILOG CPLEX is active; the best way to do this is to* `malloc` *the string.*

CPXputenv Routine

```
char *instr = NULL;
char *envstr  = NULL;

/* Initialize the CPLEX environment */

envstr = (char *) malloc (256);
if ( envstr == NULL ) {
   fprintf (stderr, "Memory allocation for CPXputenv failed.\n");
   status = FAIL;
   goto TERMINATE;
}
else {
   instr = (char *)  getenv("MYAPP_HOME");
   if ( instr == NULL ) {
      fprintf (stderr, "Unable to find installation directory.\n");
      status = FAIL;
      goto TERMINATE;
   }
   strcpy (envstr, "ILOG_LICENSE_FILE=");
   strcat (envstr, instr);
   strcat (envstr, "\\license\\access.ilm");
   CPXputenv (envstr);
}

env = CPXopenCPLEX (&status);
```

The putenv Routine for Java Users

Here is an example using Concert Technology for Java users:

```
IloCplex.putenv("ILOG_LICENSE_FILE=\\license\\access.ilm");
try {
   cplex = new IloCplex();
}
catch (IloException e) {
   System.err.println("Exception caught for runtime license:" + e);
}
```

CPXRegisterLicense Routine

The following is an example illustrating the usage of the `CPXRegisterLicense()` routine.

```
static char *ilm_license=\
 "LICENSE ILOG Incline\n\
  RUNTIME CPLEX      8.000 21-Apr-2002 R81GM34ECZTS N IRIX , options: m ";
static int ilm_license_signature=2756133;

   CPXENVptr     env = NULL;
   int           status;

   /* Initialize the CPLEX environment */

    status = CPXRegisterLicense (ilm_license, ilm_license_signature);
    if ( status != 0) {
       fprintf (stderr, "Could not register CPLEX license, status %d.\n",
                status);
       goto TERMINATE;
    }
    env = CPXopenCPLEX (&status);
    if ( env == NULL ) {
       char  errmsg[1024];
       fprintf (stderr, "Could not open CPLEX environment.\n");
       CPXgeterrorstring (env, status, errmsg);
       fprintf (stderr, "%s", errmsg);
       goto TERMINATE;
    }
```

The registerLicense Routine for Java Users

Here is an example using `IloCplex.registerLicense`, for Java users:

```
static String ilm_CPLEX_license=
"LICENSE ILOG Test\n RUNTIME CPLEX      8.000 021-Apr-2002 R81GM34ECZTS N IRIX ,
options: m ";
static int ilm_CPLEX_license_signature=2756133;

public static void main(String[] args) {
```

```
   try {
      IloCplex.registerLicense(ilm_CPLEX_license, ilm_CPLEX_license_signature);
      IloCplex cplex = new IloCplex();
   }
   catch (IloException e) {
      System.err.println("Exception caught for runtime license:" + e);
   }
}
```

Summary

ILM runtime license keys come in two forms. Users of file-based `RTNODE`, `RTSTOKEN` and `TOKEN` keys should use the `CPXputenv()` or `IloCplex.putenv()` routine to identify the location `CPXputenv()`. Users of memory-based `RUNTIME` keys should use the `CPXRegisterLicense()` or `IloCplex.registerLicense()` routine to pass the key to the ILOG License Manager embedded inside the CPLEX Component Libraries. Refer to the *ILOG License Manager* online documentation that comes with your software for additional information on activating and maintaining ILM license keys.

A

ILOG CPLEX Parameter Names

In the parts of this manual where the algorithmic discussion is independent of the compiler or language, CPLEX parameter names are generally given in terms of their Concert naming convention. For convenience, Table A.1, *Parameter Name Equivalents* in this appendix lists the correspondence of these names to those used by the Callable Library and the Interactive Optimizer. For details of each of these parameters, consult Appendix A of the *ILOG CPLEX Reference Manual*.

In most cases, parameter names follow these conventions:

Concert Technology in C++ Programs

In C++ programs using Concert, the parameter name is preceded by "`IloCplex`::". It is not generally necessary to distinguish among integer, Boolean, numeric, and string parameters. Example usage:

```
cplex.setParam(IloCplex::AdvInd, 0);
```

Concert Technology in Java Programs

In Java programs using Concert, the parameter name is preceded by "`IloCplex`.", then either "`IntParam`." or "`NumParam`." "`BooleanParam`.", or "`StringParam`.", depending on the entry shown in the Type column of the table. Example usage:

```
cplex.setParam(IloCplex.BooleanParam.AdvInd, false);
```

Callable Library

The Callable Library puts the parameter name in all capital letters, preceded by "`CPX_PARAM_`". Parameters are accessed by functions such as `CPXsetintparam()`,

CPXsetdblparam(), and CPXsetstrparam(). Boolean parameters are treated the same as integer parameters in the Callable Library. Example usage:

```
status = CPXsetintparam (env, CPX_PARAM_ADVIND, CPX_OFF);
```

Interactive Optimizer

The Interactive Optimizer user sets parameters via a menu or submenu, usually with a similar but not identical name to the Concert name. Example usage:

```
CPLEX> SET ADVANCE NO
```

Exceptions

Exceptions are marked in the manual with an asterisk. They are:

- PreLinear is not available in Concert, as its functionality is handled by other means such as callbacks.
- RootAlg in Concert has multiple meanings in the other two interfaces.
- ScrInd is available only in the Callable Library, as its functionality is limited to this interface.

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
AdvInd	Bool	CPX_PARAM_ADVIND	ADVANCE
AggCutLim	Int	CPX_PARAM_AGGCUTLIM	MIP LIMITS AGGFORCUT
AggFill	Int	CPX_PARAM_AGGFILL	PREPROCESSING FILL
AggInd	Int	CPX_PARAM_AGGIND	PREPROCESSING AGGREGATOR
BarAlg	Int	CPX_PARAM_BARALG	BARRIER ALGORITHM
BarColNz	Int	CPX_PARAM_BARCOLNZ	BARRIER COLNONZEROS
BarCrossAlg	Int	CPX_PARAM_BARCROSSALG	BARRIER CROSSOVER
BarDisplay	Int	CPX_PARAM_BARDISPLAY	BARRIER DISPLAY
BarEpComp	Num	CPX_PARAM_BAREPCOMP	BARRIER CONVERGETOL
BarGrowth	Num	CPX_PARAM_BARGROWTH	BARRIER LIMITS GROWTH
BarItLim	Int	CPX_PARAM_BARITLIM	BARRIER LIMITS ITERATIONS
BarMaxCor	Int	CPX_PARAM_BARMAXCOR	BARRIER LIMITS CORRECTIONS
BarObjRng	Num	CPX_PARAM_BAROBJRNG	BARRIER LIMITS OBJRANGE

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
BarOOC	Bool	CPX_PARAM_BAROOC	BARRIER OUTOFCORE
BarOrder	Int	CPX_PARAM_BARORDER	BARRIER ORDERING
BarStartAlg	Int	CPX_PARAM_BARSTARTALG	BARRIER STARTALG
BarThreads	Int	CPX_PARAM_BARTHREADS	BARRIER LIMITS THREADS
BarVarUp	Num	CPX_PARAM_BARVARUP	BARRIER LIMITS VARUPPER
BasInterval	Int	CPX_PARAM_BASINTERVAL	SIMPLEX BASISINTERVAL
BBInterval	Int	CPX_PARAM_BBINTERVAL	MIP STRATEGY BBINTERVAL
BndStrenInd	Int	CPX_PARAM_BNDSTRENIND	PREPROCESSING BOUNDSTRENGTH
BrDir	Int	CPX_PARAM_BRDIR	MIP STRATEGY BRANCH
BtTol	Num	CPX_PARAM_BTTOL	MIP STRATEGY BACKTRACK
Cliques	Int	CPX_PARAM_CLIQUES	MIP CUTS CLIQUES
ClockType	Int	CPX_PARAM_CLOCKTYPE	CLOCKTYPE
CoeRedInd	Int	CPX_PARAM_COEREDIND	PREPROCESSING COEFFREDUCE
ColGrowth	Int	CPX_PARAM_COLGROWTH	READ VARIABLES
Covers	Int	CPX_PARAM_COVERS	MIP CUTS COVERS
CraInd	Int	CPX_PARAM_CRAIND	SIMPLEX CRASH
CutLo	Num	CPX_PARAM_CUTLO	MIP TOLERANCES LOWERCUTOFF
CutPass	Int	CPX_PARAM_CUTPASS	MIP LIMITS CUTPASSES
CutsFactor	Num	CPX_PARAM_CUTSFACTOR	MIP LIMITS CUTSFACTOR
CutUp	Num	CPX_PARAM_CUTUP	MIP TOLERANCES UPPERCUTOFF
DataCheck	Bool	CPX_PARAM_DATACHECK	READ DATACHECK
DepInd	Bool	CPX_PARAM_DEPIND	PREPROCESSING DEPENDENCY
DisjCuts	Int	CPX_PARAM_DISJCUTS	MIP CUTS DISJUNCTIVE
DiveType	Int	CPX_PARAM_DIVETYPE	MIP STRATEGY DIVE
DPriInd	Int	CPX_PARAM_DPRIIND	SIMPLEX DGRADIENT

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
EpAGap	Num	CPX_PARAM_EPAGAP	MIP TOLERANCES ABSMIPGAP
EpGap	Num	CPX_PARAM_EPGAP	MIP TOLERANCES MIPGAP
EpInt	Num	CPX_PARAM_EPINT	MIP TOLERANCES INTEGRALITY
EpMrk	Num	CPX_PARAM_EPMRK	SIMPLEX TOLERANCES MARKOWITZ
EpOpt	Num	CPX_PARAM_EPOPT	SIMPLEX TOLERANCES OPTIMALITY
EpPer	Num	CPX_PARAM_EPPER	SIMPLEX PERTURBATION
EpRHS	Num	CPX_PARAM_EPRHS	SIMPLEX TOLERANCES FEASIBILITY
FlowCovers	Int	CPX_PARAM_FLOWCOVERS	MIP CUTS FLOWCUTS
FlowPaths	Int	CPX_PARAM_FLOWPATHS	MIP CUTS PATHCUT
FracCand	Int	CPX_PARAM_FRACCAND	MIP LIMITS GOMORYCAND
FracCuts	Int	CPX_PARAM_FRACCUTS	MIP CUTS GOMORY
FracPass	Int	CPX_PARAM_FRACPASS	MIP LIMITS GOMORYPASS
GUBCovers	Int	CPX_PARAM_GUBCOVERS	MIP CUTS GUBCOVERS
HeurFreq	Int	CPX_PARAM_HEURFREQ	MIP STRATEGY HEURISTICFREQ
IISInd	Int	CPX_PARAM_IISIND	SIMPLEX IISFIND
ImplBd	Int	CPX_PARAM_IMPLBD	MIP CUTS IMPLIED
IntSolLim	Int	CPX_PARAM_INTSOLLIM	MIP LIMITS SOLUTIONS
ItLim	Int	CPX_PARAM_ITLIM	SIMPLEX LIMITS ITERATIONS
MIPDisplay	Int	CPX_PARAM_MIPDISPLAY	MIP DISPLAY
MIPEmphasis	Int	CPX_PARAM_MIPEMPHASIS	MIP EMPHASIS
MIPInterval	Int	CPX_PARAM_MIPINTERVAL	MIP INTERVAL
MIPOrdInd	Bool	CPX_PARAM_MIPORDIND	MIP STRATEGY ORDER
MIPOrdType	Int	CPX_PARAM_MIPORDTYPE	MIP ORDERTYPE
MIPStart	Bool	CPX_PARAM_MIPSTART	MIP STRATEGY MIPSTART
MIPThreads	Int	CPX_PARAM_MIPTHREADS	MIP LIMITS THREADS

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
MIRCuts	Int	CPX_PARAM_MIRCUTS	MIP CUTS MIRCUT
NetDisplay	Int	CPX_PARAM_NETDISPLAY	NETWORK DISPLAY
NetEpOpt	Num	CPX_PARAM_NETEPOPT	NETWORK TOLERANCES OPTIMALITY
NetEpRHS	Num	CPX_PARAM_NETEPRHS	NETWORK TOLERANCES FEASIBILITY
NetFind	Int	CPX_PARAM_NETFIND	NETWORK NETFIND
NetItLim	Int	CPX_PARAM_NETITLIM	NETWORK ITERATIONS
NetPPriInd	Int	CPX_PARAM_NETPPRIIND	NETWORK PRICING
NodeAlg	Int	CPX_PARAM_SUBALG	MIP STRATEGY SUBALGORITHM
NodeFileInd	Int	CPX_PARAM_NODEFILEIND	MIP STRATEGY FILE
NodeLim	Int	CPX_PARAM_NODELIM	MIP LIMITS NODES
NodeSel	Int	CPX_PARAM_NODESEL	MIP STRATEGY NODESELECT
NzGrowth	Int	CPX_PARAM_NZGROWTH	READ NONZEROS
ObjDif	Num	CPX_PARAM_OBJDIF	MIP TOLERANCES OBJDIFFERENCE
ObjLLim	Num	CPX_PARAM_OBJLLIM	SIMPLEX LIMITS LOWEROBJ
ObjULim	Num	CPX_PARAM_OBJULIM	SIMPLEX LIMITS UPPEROBJ
PerInd	Bool	CPX_PARAM_PERIND	SIMPLEX PERTURBATION
PerLim	Int	CPX_PARAM_PERLIM	SIMPLEX LIMITS PERTURBATION
PPriInd	Int	CPX_PARAM_PPRIIND	SIMPLEX PGRADIENT
PreCompress	Int	CPX_PARAM_PRECOMPRESS	PREPROCESSING COMPRESS
PreDual	Int	CPX_PARAM_PREDUAL	PREPROCESSING DUAL
PreInd	Bool	CPX_PARAM_PREIND	PREPROCESSING PRESOLVE
PreLinear*	Int	CPX_PARAM_PRELINEAR	PREPROCESSING LINEAR
PrePass	Int	CPX_PARAM_PREPASS	PREPROCESSING NUMPASS
PreslvNd	Int	CPX_PARAM_PRESLVND	MIP STRATEGY PRESOLVENODE
PriceLim	Int	CPX_PARAM_PRICELIM	SIMPLEX PRICING

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
Probe	Int	CPX_PARAM_PROBE	MIP STRATEGY PROBE
QPmakePSDInd	Bool	CPX_PARAM_QPMAKEPSDIND	PREPROCESSING QPMAKEPSD
QPNzGrowth	Int	CPX_PARAM_QPNZGROWTH	READ QPNONZEROS
Reduce	Int	CPX_PARAM_REDUCE	PREPROCESSING REDUCE
ReInv	Int	CPX_PARAM_REINV	SIMPLEX REFACTOR
RelaxPreInd	Bool	CPX_PARAM_RELAXPREIND	PREPROCESSING RELAX
RelObjDif	Num	CPX_PARAM_RELOBJDIF	MIP TOLERANCES RELOBJDIFFERENCE
ReverseInd	Bool	CPX_PARAM_REVERSEIND	READ REVERSE
RootAlg*	Int	CPX_PARAM_LPMETHOD	LPMETHOD
RootAlg*	Int	CPX_PARAM_QPMETHOD	QPMETHOD
RootAlg*	Int	CPX_PARAM_STARTALG	MIP STRATEGY STARTALGORITHM
RowGrowth	Int	CPX_PARAM_ROWGROWTH	READ CONSTRAINTS
ScaInd	Int	CPX_PARAM_SCAIND	READ SCALE
ScrInd*	Int	CPX_PARAM_SCRIND	
SiftAlg	Int	CPX_PARAM_SIFTALG	SIFTING ALGORITHM
SiftDisplay	Int	CPX_PARAM_SIFTDISPLAY	SIFTING DISPLAY
SiftItLim	Int	CPX_PARAM_SIFTITLIM	SIFTING ITERATIONS
SimDisplay	Int	CPX_PARAM_SIMDISPLAY	SIMPLEX DISPLAY
SimThreads	Int	CPX_PARAM_SIMTHREADS	SIMPLEX LIMITS THREADS
SingLim	Int	CPX_PARAM_SINGLIM	SIMPLEX LIMITS SINGULARITY
StrongCandLim	Int	CPX_PARAM_STRONGCANDLIM	MIP LIMITS STRONGCAND
StrongItLim	Int	CPX_PARAM_STRONGITLIM	MIP LIMITS STRONGIT
StrongThreadLim	Int	CPX_PARAM_STRONGTHREADLIM	MIP LIMITS STRONGTHREADS
Symmetry	Bool	CPX_PARAM_SYMMETRY	PREPROCESSING SYMMETRY
Threads	Int	CPX_PARAM_THREADS	THREADS

Table A.1 *Parameter Name Equivalents*

Concert	Type	Callable Library	Interactive Optimizer
TiLim	Num	CPX_PARAM_TILIM	TIMELIMIT
TreLim	Num	CPX_PARAM_TRELIM	MIP LIMITS TREEMEMORY
VarSel	Int	CPX_PARAM_VARSEL	MIP STRATEGY VARIABLESELECT
WorkDir	Str	CPX_PARAM_WORKDIR	WORKDIR
WorkMem	Num	CPX_PARAM_WORKMEM	WORKMEM
XXXInd	Bool	CPX_PARAM_XXXIND	SIMPLEX XXXSTART

B

Interactive Optimizer Commands

The following table lists Interactive Optimizer commands, their primary options, and pages in this manual on which usage examples can be found.

Command		Options	Example
add			
baropt			page 374
baropt	dualopt		
baropt	primopt		
baropt	stop		
change	bounds		
change	coefficient		
change	delete		
change	name		
change	objective		

Command		Options	Example
change	problem	*type*	page 243, page 198, page 202
change	qpterm		page 200
change	rhs		
change	sense		
change	type		page 228
display	iis		page 143
display	problem	all	
display	problem	binaries	page 226
display	problem	bounds	
display	problem	constraints	
display	problem	generals	page 226
display	problem	histogram	page 167
display	problem	integers	page 226
display	problem	names	
display	problem	qpvariables	
display	problem	semi-continuous	
display	problem	sos	
display	problem	stats	page 121, page 226
display	problem	variable	
display	sensitivity		
display	settings		
display	settings	all	
display	settings	changed	
display	solution	basis	

Command		Options	Example
display	solution	bestbound	
display	solution	dual	
display	solution	kappa	page 137
display	solution	objective	
display	solution	quality	page 141, page 163
display	solution	reduced	
display	solution	slacks	
display	solution	variables	
enter			
help			
mipopt			page 229
netopt			page 178, page 258
optimize			
primopt			page 374
quit			
read	*filename*	*type*	page 130, page 178, page 130
set	advance	yes	page 130
set	barrier		page 159
set	barrier	algorithm	page 163, page 170
set	barrier	colnonzeros	page 171
set	barrier	convergetol	
set	barrier	crossover	
set	barrier	display	page 160, page 171

Command		Options	Example
set	barrier limits	corrections	page 170
set	barrier limits	growth	
set	barrier limits	iterations	
set	barrier limits	objrange	page 172
set	barrier limits	threads	
set	barrier limits	varupper	
set	barrier	ordering	
set	barrier	outofcore	
set	barrier	startalg	
set	clocktype		
set	defaults		
set	logfile	*filename*	
set	lpmethod		
set	mip cuts	*class*	page 237
set	mip	display	
set	mip	emphasis	
set	mip	interval	
set	mip limits	aggforcut	
set	mip limits	cutpasses	
set	mip limits	cutsfactor	page 237
set	mip limits	gomorycand	
set	mip limits	gomorypass	
set	mip limits	nodes	page 242
set	mip limits	solutions	page 242
set	mip limits	strongcand	
set	mip limits	strongit	

Command		Options	Example
set	mip limits	strongthreads	
set	mip limits	treememory	page 254
set	mip	ordtype	
set	mip strategy	backtrack	page 232
set	mip strategy	bbinterval	page 232
set	mip strategy	branch	page 232
set	mip strategy	dive	
set	mip strategy	file	
set	mip strategy	heuristicfreq	page 238
set	mip strategy	mipstart	page 240
set	mip strategy	nodeselect	page 232
set	mip strategy	order	
set	mip strategy	presolvenode	
set	mip strategy	probe	page 238
set	mip strategy	startalgorithm	
set	mip strategy	subalgorithm	page 258
set	mip strategy	variableselect	page 232
set	mip tolerances	absmipgap	
set	mip tolerances	integrality	
set	mip tolerances	lowercutoff	
set	mip tolerances	mipgap	
set	mip tolerances	objdifference	page 252
set	mip tolerances	relobjdifference	
set	mip tolerances	uppercutoff	
set	network	display	page 179
set	network	iterations	page 179

Command		Options	Example
set	network	netfind	
set	network	pricing	page 179
set	network tolerances	feasibility	page 179
set	network tolerances	optimality	
set	output	*channel*	
set	output	logonly	page 141, page 143
set	preprocessing	aggregator	page 127, page 135, page 239
set	preprocessing	boundstrength	page 239
set	preprocessing	coeffreduce	page 239, page 258
set	preprocessing	compress	
set	preprocessing	dependency	page 127, page 167
set	preprocessing	dual	page 168
set	preprocessing	fill	page 127
set	preprocessing	linear	
set	preprocessing	numpass	
set	preprocessing	presolve	page 135, page 239
set	preprocessing	qpmakepsd	
set	preprocessing	reduce	page 127, page 140
set	preprocessing	relax	page 239
set	preprocessing	symmetry	
set	qpmethod		
set	read	constraints	

Command		Options	Example
set	read	datacheck	
set	read	nonzeroes	
set	read	qpnonzeroes	
set	read	reverse	
set	read	scale	page 132
set	read	variables	
set	sifting	algorithm	
set	sifting	display	
set	sifting	iterations	
set	simplex	basisinterval	
set	simplex	crash	page 133
set	simplex	dgradient	
set	simplex	display	
set	simplex	iisfind	
set	simplex limits	iterations	
set	simplex limits	lowerobj	
set	simplex limits	perturbation	page 138
set	simplex limits	singularity	page 138
set	simplex limits	threads	
set	simplex limits	upperobj	
set	simplex	perturbation	page 138
set	simplex	pgradient	
set	simplex	pricing	
set	simplex	refactor	page 134
set	simplex tolerances	feasibility	page 142
set	simplex tolerances	markowitz	page 139

Command		Options	Example
set	simplex tolerances	optimality	page 142
set	simplex	xxxstart	
set	threads		
set	timelimit		page 242
set	workdir	*prompt for new directory*	
set	workmem	*prompt for new value of working memory available*	page 242
tranopt			
write	*filename*	*type*	page 127, page 138, page 145, page 257, page 198
xecute	command		page 145

Managing Parameters in the Interactive Optimizer

To see the current value of a parameter that interests you in the Interactive Optimizer, use the command `display settings`. The command `display settings changed` lists only those parameters whose values are not the default value. The command `display settings all` lists all parameters and their values.

To change the value of a parameter in the Interactive Optimizer, use the command `set` followed by options to indicate the parameter and the value you want it to assume.

The *ILOG CPLEX Reference Manual* documents the name of each parameter and its options in the Interactive Optimizer.

Saving a Parameter Specification File

You can tell the ILOG CPLEX Interactive Optimizer to read customized parameter settings from a *parameter specification file*. By default, ILOG CPLEX expects a parameter specification file to be named `cplex.par`, and it looks for that file in the directory where it

is executing. However, you can rename the file, or tell ILOG CPLEX to look for it in another directory by setting the system environment variable `CPLEXPARFILE` to the full path name to your parameter specification file. You set that environment variable in the customary way for your platform. For example, on a Unix platform, you might use a shell command to set the environment variable, or on a personal computer running NT, you might click on the System icon in the control panel, then select the environment tab from the available system properties tabs.

During initialization in the Interactive Optimizer, ILOG CPLEX locates any available parameter specification file (by checking the current execution directory for `cplex.par` and by checking the environment variable `CPLEXPARFILE`) and reads that file. As it opens the file, ILOG CPLEX displays the message "Initial parameter values are being read from `cplex.par`" (or from the parameter specification file you specified). As ILOG CPLEX displays that message on the screen, it also writes the message to the log file. If ILOG CPLEX cannot open the file, it displays no message, records no note in the log file, and uses default parameter settings.

You can use a parameter specification file to change any parameter or parameters accessible by the `set` command in the Interactive Optimizer. The parameter types, names, and options are those used by the `set` command in the Interactive Optimizer.

To create a parameter specification file, you can use either of these alternatives:

- Use an ordinary text editor to create a file where each line observes the following syntax:

  parameter-name option value

- Use the command `display settings` in the Interactive Optimizer to generate a list of current parameter settings. Those settings will be recorded in the log file. You can then edit the log file to create your parameter specification file.

  `display settings changed` lists parameters different from the default.

  `display settings all` lists all parameters.

Each entry on a line must be separated by at least one space or tab. Blank lines in a parameter specification file are acceptable; there are no provisions for comments in the file. You may abbreviate parameter names to unique character sequences, as you do in the `set` command.

As ILOG CPLEX reads a parameter specification file, if the parameter name and value are valid, ILOG CPLEX sets the parameter and writes a message about it to the screen and to the log file. If ILOG CPLEX encounters a repeated parameter, it uses the last value specified. ILOG CPLEX terminates under the following conditions:

- if it encounters a parameter that is unknown;
- if it encounters a parameter that is not unique;
- if the parameter is correctly specified but the value is missing, invalid, or out of range.

Here is an example of a parameter specification file that resets the limits on the size of problem reads and opens a log file named `problem.log`.

```
read constraints    50
read variables     100
read nonzeros      500
logfile            problem.log
```

Index

A

absolute objective difference **252**
absolute optimality tolerance **252**
accessing
  basis information **43**
  current parameter value **40**, **98**
  default parameter value **98**
  dual values **43**
  maximum parameter value **98**
  minimum parameter value **98**
  objective function value **43**
  parameters in Interactive Optimizer **400**
  reduced costs **43**
  slack values **43**
  solution quality **45**
  solution values **43**
active model **64**
  MIP **69**
active node **319**
`add` member function
  `IloModel` class **36**, **45**
`addMinimize` **66**
advanced basis
  in networks **126**
  reading **130**
  saving **130**
  starting from **129**
  starting parameter **130**
advanced starting basis **148**
`AggCutLim` **238**
aggregator
  barrier preprocessing **167**
`alg` **69**
algorithm
  choosing in LP **39**
  controlling in `IloCplex` **40**
  pricing **130**
  type **70**
`Algorithm.Barrier` **72**
application
  creating with Concert Technology **33**
  development steps **33**
arc **176**
architecture **83**
arguments
  null pointers **92**
  optional **92**
array
  constructing **76**
  creating multi-dimensional **50**
  extensible **36**
  using for I/O **51**
aspect ratio **379**

B

barrier **72**
barrier optimizer
  algorithm **157**

algorithms and infeasibility **172**
barrier display parameter **170**
centering corrections **170**
column nonzeros parameter **167**, **171**
corrections limit **170**
growth parameter **171**
infeasibility analysis **172**
linear **156** to **173**
log file **160**
numerical difficulties and **170**
out-of-core barrier **166**
parallel **379**
performance tuning **165**
preprocessing **167**
primal-dual **126**
quadratic **195** to **221**
row-ordering algorithms **167**
simplex optimizer and **158**
solution quality **163**
solving LP problems **156**
starting-point heuristics **168**
unbounded optimal face and **171**
unbounded problems **172**
uses **158**
working storage allocation **166**
BarThreads **376**
BAS file format **130**, **138**, **290**
basis **73**
accessing information **43**
condition number **137**, **141**
crossover algorithms **158**
current **85**
file formats to save **290**, **291**
previous optimal **87**
removing objects from **46**
saving best so far **138**
unstable optimal **141**
see also advanced basis
Best Bound **233**
Best Estimate **233**
bibliography **30**
BIN file format **290**
Boolean or semi-continuous variables
single LP **69**
`boolVar` **62**
`boolVarArray` **62**
bound
violation **142**
`Bounded`
return status **67**
branch & cut **69**
algorithm **231**
parallel processing **375**
parameters **232**
solves MIP models **229**
tree **231**
branch & cut algorithm
memory problems and **253**
priority order **240**
special ordered sets (SOS) **244**
storing tree nodes **254**
tree subproblems **257**
branching directions **72**
`BtTol` **233**

C

C++ parameter names **385**
Calable Library parameter names **385**
call by value **88**
Callable Library
categories of routines **84**
core **84**
debugging and **118**
description **21**
parallel optimizers **374**
parameters **98**
using **83** to **111**
see also individual CPX*xxx* routines
callback
graphic user interface and **352**
resetting to null **41**, **99**
using status variables **352**
changing
limit on barrier corrections **170**
maximization to minimization **182**
pricing algorithm **179**
problem type
network to LP **189**
qp **200**

zeroed_qp **200**
quadratic coefficients **199**, **200**
variable type **47**
channel
example **298**
character string
length requirements **93**
`check.c` CPLEX file **94**
Cholesky factor **166**
barrier iteration **158**
barrier log file and **162**
choosing
algorithm in LP **39**
clique cuts
defined **235**
`cloneK.log` **379**
clones **379**
log files **379**
threads and **379**
closing
application **87**, **182**
environment **87**, **182**
log files **295**
`col.and()` **77**
column
dense **171**
density **167**
growth parameter **182**
index number **92**
modeling by **75**
name **92**
nonzeros parameter **167**, **171**
referencing **92**
columnwise modeling **50**, **61**, **75**, **100**
complementarity **157**, **158**, **172**
convergence tolerance **171**
Component Libraries
defined **22**
Concert Technology Library
accessing parameter values **40**
application development steps **33**
creating application **33**
description **21**
design **31**
error handling **47**
solving problem with **31**
using **31** to **58**
writing programs with **31**
see also individual Ilo*xxx* routines
constraint
accessing slack values **43**
cumulative **146**, **147**
cuts as **235**
modeling linear **37**
removing from basis **46**
representing with `IloRange` **35**
violation **142**
constraints
create ranged **61**
ranged **63**
constructing arrays of variables **76**
continuous relaxation **231**
continuous relaxation subproblem **319**
conventions
character strings **93**
in parameter specification file **401**
naming **293**
notation **26**
numbering **121**, **292**
numbering rows, columns **120**
`convert` CPLEX utility **294**
converting
error code to string **181**
file formats **294**
network-flow model to LP **188**
network-flow problem to LP **189**
convex quadratic programming *see* quadratic programming

copying
variable types in MIP **90**
variable types in QP **90**
cover cuts **235**
defined **235**
CPLEX
Component Libraries **22**
core **84**
licensing **32**
parameters **40**
`cplex` object
creating **38**

notifying changes to **45**
cplex.h header file **95**, **97**, **98**, **120**
cplex.log file
changing name **160**
clone logs **379**
default name **131**, **295**
cplex.par parameter specification file **400**
CPLEXPARFILE environment variable **401**
CPX_ALG_AUTO symbolic constant **258**
CPX_ALG_DUAL symbolic constant **258**
CPX_ALG_HYBNETOPT symbolic constant **258**
CPX_ALG_PRIMAL symbolic constant **258**
CPX_PARAM_ADVIND parameter **130**
CPX_PARAM_AGGIND parameter **239**
CPX_PARAM_BARSTARTALG parameter **169**
CPX_PARAM_BARTHREADS parameter **371**
CPX_PARAM_BBINTERVAL parameter **232**
CPX_PARAM_BNDSTRENIND parameter **239**
CPX_PARAM_BRDIR parameter **232**
CPX_PARAM_BTTOL parameter **232**
CPX_PARAM_CLIQUES parameter **237**
CPX_PARAM_CLOCKTYPE parameter **98**, **374**
CPX_PARAM_COEREDIND parameter **239**
CPX_PARAM_COLGROWTH parameter **90**, **182**
CPX_PARAM_COVERS parameter **237**
CPX_PARAM_DISJCUTS parameter **237**
CPX_PARAM_FLOWCOVERS parameter **237**
CPX_PARAM_FLOWPATHS parameter **237**
CPX_PARAM_FRACCUTS parameter **237**
CPX_PARAM_GUBCOVERS parameter **237**
CPX_PARAM_IMPLBD parameter **237**
CPX_PARAM_INTSOLLIM parameter **242**
CPX_PARAM_MIPTHREADS parameter **372**
CPX_PARAM_MIRCUTS parameter **237**
CPX_PARAM_NODELIM parameter **242**
CPX_PARAM_NODESEL parameter **232**
CPX_PARAM_NZGROWTH parameter **90**
CPX_PARAM_PREIND parameter **239**
CPX_PARAM_PROBE parameter **238**
CPX_PARAM_RELAXPREIND parameter **239**
CPX_PARAM_ROWGROWTH parameter **90**, **182**
CPX_PARAM_SCRIND parameter **94**, **120**, **138**, **151**, **181**, **297**, **351**
CPX_PARAM_SIMTHREADS parameter **372**
CPX_PARAM_STARTALG parameter **257**
CPX_PARAM_STRONGTHREADLIM parameter **372**
CPX_PARAM_THREADS parameter **371**
CPX_PARAM_TILIM parameter **242**
CPX_PARAM_TRELIM parameter **242**
CPX_PARAM_VARSEL parameter **232**
CPX_PARAM_WORKMEM parameter **242**
CPX_SEMICONT **246**
CPX_SEMIINT **246**
CPXaddchannel routine **89**, **298**
CPXaddcols routine **86**, **90**, **114**
CPXaddfpdest routine **95**, **298**, **299**
CPXaddfuncdest routine **95**, **298**, **299**
CPXaddrows routine **86**, **90**, **101**, **114**
CPXALG_BARRIER symbolic constant **258**
CPXCENVptr **89**
CPXCHANNELptr data type **89**
CPXCHARptr data type **95**
CPXcheckaddcols routine **94**
CPXcheckaddrows routine **94**
CPXcheckchgcoeflist routine **94**
CPXcheckcopyctype routine **94**
CPXcheckcopylp routine **94**
CPXcheckcopylpwnames routine **94**
CPXcheckcopyqsep routine **94**
CPXcheckcopyquad routine **94**
CPXcheckcopysos routine **94**
CPXcheckvals routine **94**
CPXchgcoeflist routine **86**, **114**
CPXchgprobtype routine **202**, **243**, **261**
CPXchgqpcoef routine **200**
CPXcloseCPLEX routine **87**, **151**, **182**, **215**, **271**, **299**, **352**
CPXCLPptr **89**
CPXCNETptr **89**
CPXcopybase routine **150**, **151**
CPXcopycttype routine **246**
CPXcopyctype routine **90**, **119**, **225**, **260**
CPXcopylp routine **86**, **114**
CPXcopynettolp routine **188**
CPXcopyorder routine **245**, **277**
CPXcopyqsep routine **90**
CPXcopyquad routine **90**, **207**
CPXcopysos routine **245**, **246**, **277**
CPXcreateprob **351**
CPXcreateprob routine **86**, **89**, **101**, **151**, **207**, **215**,

260, **271**
CPXdelchannel routine **298**, **299**
CPXdelfpdest routine **95**, **298**, **299**
CPXdelfuncdest routine **298**, **299**
CPXdisconnectchannel routine **298**
CPXdisplayiis routine **144**
CPXdualopt routine **379**
CPXENVptr data type **89**
CPXERR_NEGATIVE_SURPLUS symbolic constant **308**
cpxerror message channel **297**, **299**
CPXfclose routine **95**
CPXFILEptr data type **95**
CPXflushchannel routine **298**
CPXfopen routine **95**, **295**
CPXfputs routine **96**
CPXfreeprob routine **87**, **151**, **215**, **271**, **352**
CPXgetcallbackinfo routine **94**, **349**, **350**, **352**
CPXgetchannels routine **89**, **297**, **299**
CPXgetcolindex routine **93**
CPXgetcolname routine **309**, **310**
CPXgetcols routine **308**, **309**
CPXgetctype routine **226**
CPXgetdblparam routine **92**, **98**
CPXgetdblquality routine **138**, **143**, **164**
CPXgeterrorstring routine **181**, **351**
CPXgetintparam routine **92**, **98**
CPXgetintquality routine **164**
CPXgetmipobjval routine **261**, **271**
CPXgetmipslack routine **261**
CPXgetmipx routine **261**, **271**
CPXgetnumcols routine **91**
CPXgetrowindex routine **93**
CPXgetrowname routine **91**
CPXgetsos routine **226**
CPXgetstat routine **261**, **271**, **350**
CPXgetstrparam routine **92**, **98**
CPXgetx routine **88**
CPXiiswrite routine **144**, **190**
CPXinfodblparam routine **92**, **98**
CPXinfointparam routine **92**, **98**
CPXinfostrparam routine **92**, **98**
cpxlog message channel **297**
CPXlpopt **215**
CPXlpopt routine **101**, **352**
CPXLPptr data type **89**
CPXmbasewrite routine **130**
CPXmemcpy routine **96**
CPXMIP_ABORT_FEAS symbolic constant **351**
CPXMIP_ABORT_INFEAS symbolic constant **351**
CPXmipopt routine **260**, **271**
CPXmsg routine **85**, **95**, **298**, **299**
CPXmsgstr routine **96**
CPXNETaddarcs routine **181**
CPXNETaddnodes routine **181**
CPXNETcheckcopynet routine **94**
CPXNETchgobjsen routine **182**
CPXNETcreateprob routine **89**, **181**
CPXNETdelnodes routine **182**
CPXNETfreeprob routine **182**
CPXNETprimopt routine **182**, **190**
CPXNETptr data type **89**
CPXNETsolution routine **182**
CPXnewcols routine **86**, **101**, **114**
CPXnewrows routine **86**, **114**
CPXopenCPLEX routine **85**, **89**, **98**, **101**, **151**, **181**, **215**, **297**, **299**
CPXordwrite routine **277**
CPXprimopt routine **97**, **261**
CPXPROB_FIXEDMILP symbolic constant **261**
CPXPUBLIC symbolic constant **95**
CPXPUBVARARGS symbolic constant **95**
CPXqpopt **215**
CPXqpopt routine **207**
CPXqpwrite routine **198**
CPXreadcopybase routine **130**
CPXreadcopyprob routine **86**, **310**
cpxresults message channel **297**
CPXsavwrite routine **118**
CPXsetdblparam routine **92**, **99**
CPXsetdefaults routine **99**
CPXsetintparam routine **92**, **98**, **120**, **151**, **181**, **257**, **374**
CPXsetlogfile routine **94**, **95**, **120**, **160**, **295**, **296**, **297**
CPXsetlpcallbackfunc routine **95**, **350**, **352**
CPXsetmipcallbackfunc routine **95**, **350**
CPXsetstrparam routine **92**, **99**
CPXsolution routine **102**, **261**, **350**
CPXsoswrite routine **277**
CPXstrcpy routine **96**
CPXstrlen routine **96**

`CPXtreewrite` routine **242**
`CPXVOIDptr` data type **95**
`cpxwarning` message channel **297**
`CPXwriteprob` routine **102**, **118**, **138**, **277**
`CPXwritesol` routine **93**
create
  modeling variables **61**
  new rows **75**
  objective function **61**
  ranged constraints **61**
creating
  application with Concert Technology **33**
  CPLEX environment **181**
  log file **295**
  network flow problem object **181**
  parameter specification file **401**
  problem object **86**
crossover
  file format **160**
  irreducibly inconsistent set (IIS) and **173**
  verifying barrier solutions **169**
cumulative constraint **146**, **147**
cuts **235**, **319**
  adding **236**
  clique **235**
  cover **235**
  disjunctive **235**
  flow cover **235**
  flow path **236**
  Gomory fractional **236**
  GUB cover **236**
  implied bound **236**
  local or global **320**
  MIR **236**
  recorded in MIP node log file **248**
  re-optimizing **236**
  what are **235**

D

data
  entering **86**
data types
  special **89**
debugging
  Callable Library and **118**
  diagnostic routines and **94**
  heap **120**
  Interactive Optimizer and **118**
  return values and **120**
defined **84**
degeneracy **138**
  dual **257**
delete goal stacks **326**
deleting
  model objects **46**
deleting nodes **326**
deleting variables **82**
dense column **167**
Depth First **233**
destroying
  CPLEX environment **87**
  nodes **182**
  problem object **87**
devex pricing **132**
diagnosing
  infeasibility in QPs **202**
  network infeasibility **190**
diagnostic routine **94**
  log file and **94**
  message channels and **94**
  redirecting output from **94**
diet model **64**
`diff` **62**
DIMACS **291**
dimensions, checking **121**
disjunctive cuts **235**
displaying
  barrier information **160**
  barrier solution quality **163**
  basis condition **137**
  bound infeasibilities **141**
  column-nonzeros parameter **171**
  infeasibilities on screen **141**, **143**, **144**
  messages **298**
  MIP information periodically **249**
  MIP solution information **243**
  network objective values **178**
  network solution information **181**
  network solution on screen **182**

optimization progress **352**
parameter settings **400**, **401**
problem dimensions **121**
problem statistics **121**
reduced-cost infeasibilities **141**
simplex solution quality **165**
solution quality **141**
solutions on screen **299**
variables **292**
DPE file format **257**, **290**
DUA file format **290**
dual feasibility **157**
dual residual **141**
dual simplex optimizer
perturbing objective function **138**
selecting **125**
stalling **138**
see also simplex
dual variable
solution data **43**
duality gap **157**

E

elapsed time for the MIP optimizer **377**
EMB file format **290**
empty goal **321**, **326**
`end` member function
`IloEnv` class **34**
`enter` Interactive Optimizer command **225**
entering **225**
LPs for barrier optimizer **159**
mixed integer programs (MIPs) **225**
network arcs **181**
network data **181**
network data from file **189**
network nodes **181**
enumeration
`Algorithm` **39**
`BasisStatus` **44**
`BoolParam` **40**
`IntParam` **40**
`NumParam` **40**
`Quality` **45**
`Status` **42**
`String Param` **40**
environment
initializing **85**
multithreaded **85**
parameter specification file **401**
releasing **87**
variable **401**
environment object
constructing **34**
`EpAGap` **252**
`EpGap` **252**
`eq` method **63**
`Error`
return status **67**
error checking
diagnostic routines for **94**
MPS file format **293**
problem dimensions **121**
error handling
in Concert Technology Library **47**
example
columnwise modeling **50**, **100**
contents of IIS file **145**
creating multi-dimensional arrays **50**
FORTRAN **96**
message handler **298**
MIP node log file **247**
MIP optimization **258**
MIP problem from file **269**
MIP with SOS and priority orders **275**
network optimization **177**
optimizing QP **203**
output channels **298**
output from infeasibility finder **144**
parameter specification file **402**
reading QP from file **215**
rowwise modeling **49**, **100**
using arrays for I/O **51**
exception
querying **120**
executing a goal **319**
`exportModel` member function
`IloCplex` class **138**
`expr` **63**
expression

building **34**
editable **62**
linear **35**
piecewise linear **35**
expressions
construct using modeling variables **61**
external variables **87**
extra rim vectors **293**

F

`FailGoal()` **320**
feasibility
analysis and barrier optimizer **172**
check **320**
dual **133**, **157**
network flows and **177**
primal **157**
progress toward **139**, **177**
`Feasible`
return status **67**
file format
converting **294**
described **290**
example of quadratic program **215**
numerically accurate **291**
file reading routine
defined **84**
file writing routine
defined **84**
flow cover cuts
defined **235**
flow path cuts
defined **236**
FORTRAN **96**, **121**
FracCand **238**
FracPass **238**
fractional cuts
defined **236**
free row **293**

G

`ge` method **63**
generalized upper bound (GUB) **236**
see also GUB
`getBasisStatus()` **72**
`getBasisStatuses` member function
`IloCplex` class **43**
`getBoundSA` **73**
`getBoundSA` member function
`IloCplex` class **44**
`getCplexStatus` member function
`IloCplex` class **43**, **44**
`getDefault` member function
`IloCplex` class **40**
`getDual` **72**
`getDual` member function
`IloCplex` class **43**
`getDuals` **72**
`getDuals` member function
`IloCplex` class **43**
`getIIS` **73**
`getIIS` member function
`IloCplex` class **44**
`getMax` **72**
`getMax` member function
`IloCplex` class **40**
`getMessage` member function
`IloException` class **120**
`getMin` **72**
`getMin` member function
`IloCplex` class **40**
`getObjSA` **73**
`getObjSA` member function
`IloCplex` class **44**
`getObjValue` member function
`IloCplex` class **43**
`getParam` **71**
`getParam` member function
`IloCplex` class **40**
`getQuality` member function
`IloCplex` class **45**, **137**, **164**
`getRangeSA` **73**
`getReducedCost` **72**
`getReducedCost` member function
`IloCplex` class **43**
`getReducedCosts` **72**
`getReducedCosts` member function
`IloCplex` class **43**

getRHSSA member function
  IloCplex class **44**
getSlack member function
  IloCplex class **43**
getSlacks member function
  IloCplex class **43**
getStatus member function
  IloCplex class **42**
  IloCplex::Exception class **120**
getStatuses member function
  IloCplex class **148**
getting
  see accessing
getValue member function
  IloCplex class **43**
getValues member function
  IloCplex class **43**
global thread limit **376**
global thread parameter **375**
global variables **87**
goal
  empty **321**
  executing **319**
goal stack **325**
goals as parameters **320**
Gomory fractional cuts
  defined **236**
gradient parameter
  primal simplex optimizer **379**
graphic user interface (GUI) **352**
GUB
  constraint **236**
  cover cuts **236**

H

head **176**
header file **120**
heap, debugging **120**
heuristic solutions **332**
heuristics
  starting point **168**

I

IIS **73**
IIS file format **290**
ill-conditioned
  basis **142**
  factors in **142**
  problem **140**
ILM
  see ILOG License Manager (ILM)
IloAdd template class **52**
IloAddable **60**, **64**
IloAlgorithm::Exception class **48**
IloAlgorithm::Status enumeration **42**
IloAnd class **37**
IloArray template class **37**
IloColumnArray **76**
IloConversion **82**
IloConversion class **34**, **37**, **47**
IloCplex **60**
  implements IloMPModeler **61**
  objects in user application **32**
IloCplex class
  exportModel member function **138**
  getBasisStatuses member function **43**
  getBoundSA member function **44**
  getCplexStatus member function **43**, **44**
  getDefault member function **40**
  getDual member function **43**
  getDuals member function **43**
  getIIS member function **44**
  getMax member function **40**
  getMin member function **40**
  getObjSA member function **44**
  getObjValue member function **43**
  getParam member function **40**
  getQuality member function **45**, **137**, **164**
  getReducedCost member function **43**
  getReducedCosts member function **43**
  getRHSSA member function **44**
  getSlack member function **43**
  getSlacks member function **43**
  getStatus member function **42**
  getStatuses member function **148**
  getValue member function **43**

getValues member function **43**
importModel member function **130**
isDualFeasible member function **43**
isPrimalFeasible member function **43**
setDefaults member function **41**
setParam member function **41**
solve member function **39**, **42**, **44**, **45**, **46**, **53**
IloCplex::AdvInd parameter **130**
IloCplex::Algorithm enumeration **39**
IloCplex::BarStartAlg **169**
IloCplex::BasisStatus enumeration **44**
IloCplex::BoolParam enumeration **40**
IloCplex::Exception class **48**, **120**
getStatus member function **120**
IloCplex::IntParam enumeration **40**
IloCplex::Kappa **137**
IloCplex::NumParam enumeration **40**
IloCplex::Quality enumeration **45**
IloCplex::Reduce **140**
IloCplex::StringParam enumeration **40**
IloCPModeler **60**
IloEnv class **34**
end member function **34**
IloException class
getMessage member function **120**
IloExpr class **34**
ILOG License Manager (ILM)
CPLEX and **32**
IloMaximize function **36**
IloMinimize function **36**, **52**
IloModel class **35**, **36**
add member function **36**, **45**
remove member function **36**, **45**
IloModel.add **82**
IloModel.delete **82**
IloModel.remove **82**
IloModeler **60**, **61**
basic modeling **61**
IloMPModeler **60**, **61**, **74**
IloNumArray class **36**
IloNumExpr **62**, **63**
IloNumVar **60**
extension of IloNumExpr **62**
IloNumVar class **34**, **37**
IloNumVarArray class **34**
IloObjective **64**, **75**
IloObjective class **35**, **37**, **52**
setExpr member function **200**
IloObjectiveSense **63**
IloObjectiveSense.Maximize **63**
IloObjectiveSense.Minimize **63**
iloqpex1.cpp example
example
iloqpex1.cpp **203**
IloRange **64**, **75**
IloRange class **35**, **37**
IloSemiContVar class **37**
IloSolution class **38**
IloSolver **60**
IloSOS1 class **37**
IloSOS2 class **37**
implied bound cuts
defined **236**
importModel member function
IloCplex class **130**
include file **120**
inconsistent constraints **143**
incumbent
node **232**
solution **232**
index number **92**
Individual optimizer parallel processing **376**
inf.getNumVar **74**
inf.getRange **74**
infeasibility
barrier optimizer and **172**
diagnosing in network flows **190**
diagnosing in QPs **202**
displaying on screen **141**
dual **169**, **172**
during preprocessing **143**
maximum bound **141**, **142**
maximum reduced-cost **141**, **142**
network flow **177**
network optimizer and **190**
norms **163**
primal **163**, **169**, **172**
ratio in barrier log file **163**
reports **139**
scaling and **140**

unscaled **140**
infeasibility finder **143**, **172**
multiple IISs **144**
network-flows and **190**
preprocessing and **148**
sample output **144**
setting time limit on **147**
`Infeasible`
return status **67**
infeasible problem
analyzing **44**
infeasible solution
information **73**
`InfeasibleOrUnbounded`
return status **67**
initializing
CPLEX environment **181**
problem object **86**, **181**
input operator **36**
instantiating
CPLEX environment **181**
problem object **86**, **181**
`int` **70**
integer variables
single LP **69**
integrality constraints **319**
Interactive Optimizer
accessing parameters in **400**
commands **385** to ??, **393** to **400**
customized parameter settings **400**
debugging and **118**
description **21**
experimenting with optimizers **116**
improving application performance **118**
setting parameters **400**
testing code in **114**
Interactive Optimizer parameter names **386**
`IntParam` **69**
`intVar` **62**
irreducibly inconsistent constraints **143**
irreducibly inconsistent set **73**
irreducibly inconsistent set (IIS) **144**
algorithms **147**
example, cumulative constraint **146**
example, network **190**
file format for **290**
network flows and **190**
sample file **145**
`isDualFeasible` member function
`IloCplex` class **43**
`isPrimalFeasible` member function
`IloCplex` class **43**

J

Java parameter names **385**

K

knapsack constraint **235**, **236**

L

`le` method **63**
license
CPLEX **32**
parallel **371**
limiting
network iterations **179**
strong branching candidate list **252**
strong branching iterations **252**
linear expression **35**
linear objective function **37**
Linear Programming (LP) problem
see LP
linear relaxation
network flow and **190**
log file
barrier optimizer **160**
Cholesky factor in **162**
clones and **379**
closing **295**
contents **131**, **163**
creating **295**
default **295**
description **295**
diagnostic routines and **94**
iteration **135**
naming **295**
network **179**

parallel MIP optimizer and **379**
parameter **295**
parameter specifications and **401**
records IISs **144**
records infeasibilities **141**
records infeasibility information **143**
records singularities **138**
relocating **295**
renaming **295**
lower bound values
choosing **63**
LP
barrier optimizer **156**
choosing algorithm **39**
network optimizer **175**
problem formulation **22**, **157**
solving **123** to **173**
LP file format
defined **290**
IISs and **145**
QPs and **198**
row, column order **292**
special considerations **292**

M

managing
log file **295**
Markowitz tolerance **138**, **139**
maximal cliques
recorded in MIP node log file **248**
maximization
concave QPs **196**
lower cutoff parameter **253**
see also optimization
`maximize` **64**
maximum bound infeasibility **142**
maximum reduced-cost infeasibility **142**
maximum row residual **142**
memory **254**
memory allocation **166**
memory leaks **34**
memory management **134**, **253**
refactoring frequency and **135**
message channel
diagnostic routines and **94**
message handler (example) **298**
MILP
definition **69**
MIN file format **291**
minimal covers
recorded in MIP node log file **248**
minimization
convex QPs **196**
see also optimization
upper cutoff parameter **253**
`minimize` **64**
MIP **223** to **287**
active model **69**
branch & cut **69**
changing variable type **228**
memory problems and **253**
optimizer **223**
problem formulation **224**
progress reports **246**
relaxation algorithm **257**
solution information **243**
subproblem algorithm **257**
subproblems **256**
supplying first integer solution **240**
terminating optimization **241**
MIP optimizer
solution information **243**
`MIPEmphasis` **229**
`MipThreads` **376**
MIQP
definition **69**
MIR
cuts **236**
Mixed Integer Programming (MIP) problem
see MIP
model
active **64**
adding objects **45**
adding submodels **36**
deleting objects **46**
extracting **38**
manipulations **61**
modifying **82**
notifying changes to `cplex` object **45**

removing objects **45**
solving **32**, **37**, **39**
solving with `IloCplex` **53**
modeling
columnwise **50**, **100**
objects **32**
rowwise **49**, **100**
modeling by column **61**, **75**
modeling variable
`IloNumVar` **61**
modelling variables
create **61**
modifying a model **82**
MPS file format
BAS files and **290**
CPLEX extensions **293**
defined **291**
DUA format **290**
proprietary information in **291**
reference row values **246**
REW **291**
saving basis **290**
saving dual **290**
saving embedded network **290**
saving modifications **294**
saving QP **198**
weights **246**
MST file format **291**
multithreaded application
needs multiple environments **85**

N

`name` **63**
namespace conflicts **88**
naming
arcs in network flow **181**
conventions **293**
log file **295**
node file **255**
nodes in network flow **181**
`negative` **62**
Nested parallel processing **376**
NET file format **182**, **291**
`NetItLim` **179**
network
converting to LP model **188**
embedded **179**
infeasibility in **177**
modeling variables **176**
problem formulation **176**, **177**
network extractor **180**
Network Flow problem
see network
network object **176**
network optimizer **126**, **175** to **181**
column growth parameter **182**
file format to save extracted network **290**
preprocessing and **181**
problem formulation **177**
row growth parameter **182**
turn off preprocessing **181**
`nexample.net` **178**
node **319**
demand **177**
from **176**
head **176**
sink **177**
source **177**
supply **177**
tail **176**
to **176**
transshipment **177**
node file **254**
`cpx` name convention **255**
node log **246**
node problem **319**
node selection strategy
best estimate **256**
depth-first search **256**
nondeterminism **373**
nonseparable **196**
notation **26**
notation in this manual **26**
notifying
changes to `cplex` object **45**
numbering conventions
C **121**
Fortran **121**
row, column order **292**

numerical difficulties **135**, **170**
barrier growth parameter **171**
basis condition number and **137**
complementarity **171**
convergence tolerance **171**
dense columns removed **171**
infeasibility and **139**
sensitivity **137**
unbounded optimal face **171**
numerical variable **37**
`numVarArray` **62**

O

`ObjDif` **232**
objective coefficients
crash parameter and **133**
modified in log file **178**
network flows and **178**
priority and **241**
objective difference
absolute **252**
relative **252**
objective function **63**
accessing value **43**
changing sense **182**
constructor **63**
create **61**
free row as **293**
in log file **178**
in MPS file format **294**
maximization **294**
maximize **36**
minimize **36**
network flows and **176**
quadratic terms in a single LP **69**
representing with `IloObjective` **35**
sign reversal in **294**
objective value
in log file **178**
network flows and **176**
object range parameter **172**
unbounded **172**
operator << **36**
operator >> **36**
`Optimal`
return status **67**
optimality
basis condition number and **137**
cutoff parameters **253**
duality and **257**
infeasibility ration **163**
normalized error and **165**
singularities and **138**
tolerance **141**, **142**, **252**
relative **252**
tolerance, absolute **252**
tolerance, relative **252**
optimization
interrupting **352**
stopping **241**, **352**
optimization problem
defining with modeling objects **32**
representing with `IloModel` **35**
optimization routine **84**
optimizer
barrier (linear) **156** to **173**
barrier (quadratic) **195** to **221**
differences between Barrier, simplex **158**
dual simplex **125**
MIP **223**
network **126**, **175** to **181**
parallel **371** to ??
primal simplex **125**
primal-dual barrier **126**
selecting **69**
optimizing
cuts **236**
ORD file format **245**, **291**
OrGoal **325**
out-of-core barrier **166**
output
channel parameter **296**
debugging and **120**
redirecting **296**
output operator **36**

P

parallel

license **371**
optimizers **371** to ??
threads **371**
Parallel Barrier Optimizer **376**
parallel MIP optimizer
output log file **377**
parallel processing
branch & cut **375**
individual optimizer **376**
nested **376**
root relaxation **376**
selected starting algorithm **376**
Parallel Simplex Optimizer **376**
parameter
accessing current value **40**, **98**
accessing default value **98**
accessing maximum value **98**
accessing minimum value **98**
advanced starting **130**
algorithmic **165**
barrier corrections **170**
Callable Library and **98**
controlling branch & cut strategy **232**
customized **400**
gradient **130**
iisfind **147**
log file **295**
netfind **180**
object range **172**
optimality cutoff **253**
output channel **296**
perturbation limit **139**
preprocessing dependency **167**
screen indicator **181**
setting **41**, **98**, **400**
setting all defaults **41**, **99**
specification file **401**
symbolic constants as **98**
tree memory **254**
Parameter names
C++ **385**
Callable Library **385**
Interactive Optimizer **386**
Java **385**
parameter routine
defined **84**
parameter specification file **400**, **401**, **402**
creating **401**
example **402**
syntax in **401**
Parameters **70**
parameters
customized **400**
max and min values for **71**
query **71**
perturbing
file format to save perturbed problem **290**, **291**
objective function **138**
variable bounds **138**
piecewise linear expression **35**
piecewise linear functions **61**
single LP **69**
populating problem object **86**, **181**
portability **95**
PPE file format **291**
PRE file format **291**
PreCompress **135**
preprocessing
barrier optimizer **167**
dense columns removed **171**
dependency parameter **167**
IIS files and **147**
infeasibility detected in **143**
infeasibility finder and **148**
MIPs **239**
network optimizer and **181**
starting-point heuristics and **168**
presolve **71**
presolve compression parameter **129**
presolver
barrier preprocessing **167**
file format to save reduced problem **291**
pricing algorithms **179**
primal feasibility **157**
primal simplex algorithm **71**
primal simplex optimizer **125**
perturbing variable bounds **138**
stalling **138**
see also simplex
primal variables **133**

primal-degenerate problem **125**
primal-dual barrier optimizer
  see barrier optimizer
`PrimalPricing` class **71**
priority **241**
  derived from set members **245**
  file format for orders **291**
  order **240**, **245**
  special ordered sets (SOSs) and **245**
priority orders **72**
probing parameter **238**
problem
  analyzing infeasible **44**
  solving with Concert Technology Library **31**
problem formulation
  barrier **157**
  dual **157**, **159**
  ill-conditioned **140**
  infeasibility reports **139**
  linear **22**
  network **177**
  network-flow **176**
  primal **157**, **159**
  removing dense columns **167**
  switching from network to LP **188**, **189**
problem modification routines **84**
problem object
  creating **86**
  destroying **87**
  freeing **87**
  initializing **86**
  instantiating **86**
  network **176**
  populating **86**, **181**
problem query routine
  defined **84**
problem type
  changing from network to LP **189**
  changing to qp **200**
  changing to zeroed_qp **200**
`prod` **62**
pruned node **319**

Q

QP
  example **203**, **215**
  problem formulation **195**
  solution example **215**
  solving **195** to **221**
QP file format **198**, **291**
QP relaxation **202**
quadratic coefficient
  changing **199**, **200**
quadratic objective functions **37**
Quadratic Programming (QP) problem
  see QP
query routine **309**

R

ranged constraints **63**
  create **61**
ranged row **293**
reading
  advanced basis **130**
  advanced basis from file **130**
  MIP problem data **269**
  MIP problem data from file **225**
  MIP tree from TRE file **242**
  network data from file **189**
  perturbed problem from DPE file **257**
  problem data from VEC file **160**
  QP problem data from file **215**
  start values from MST file **240**
redirecting
  diagnostic routines **94**
  log file output **296**
  oputput **120**
  screen output **296**
reduced costs **72**
  accessing **43**
reduced-cost pricing **131**
reference counting **326**
reference row values **246**
reflection scaling **180**
relative objective difference **252**
relative optimality tolerance **252**

relaxation
  algorithm applied to **257**
  of MIP problem **231**
  QP **202**
RelObjDif **232**
relocating
  log file **295**
`remove` member function
  `IloModel` class **36**, **45**
renaming
  log file **295**
residual
  dual **141**
  maximum dual **142**
  maximum row **142**
  row **141**
return status
  `Bounded` **67**
  `error` **67**
  `Feasible` **67**
  `Infeasible` **67**
  `InfeasibleOrUnbounded` **67**
  `Optimal` **67**
  `Unbounded` **67**
  `Unknown` **67**
return value **91**
  debugging with **120**
  routines to access parameters **98**
REW file format **291**
RHS
  see right-hand side
right-hand side (RHS)
  file formats for **293**
rim vectors **293**
root relaxation
  parallel processing **376**
`RootAlg` **69**
row
  growth parameter **182**
  index number **92**
  name **92**
  referencing **92**
  residual **141**
row-ordering algorithms **167**
  approximate minimum degree (AMD) **167**
  approximate minimum fill (AMF) **168**
  automatic **167**
  nested dissection (ND) **168**
rowwise modeling **49**, **100**

S

SAV file format **130**, **198**, **291**
saving
  advanced basis **130**
  best factorable basis **138**
  DPE file **257**
  MIP tree **242**
  parameter specification file **400**
  perturbed problem **257**
  SAV file **257**
  TRE file **242**
scaling **140**, **142**
  alternative methods of **132**
  definition **132**
  in network extraction **180**
  infeasibility and **140**
  singularities and **138**
search tree **319**
semi-continuous variable **37**, **246**
semi-continuous variables **61**
sensitivity analysis **73**
  MIPs **243**
  performing **44**
separable **196**
`set` Interactive Optimizer command **400**
`setDefaults` **71**
`setDefaults` member function
  `IloCplex` class **41**
`setDirection` **72**
`setDirections` **72**
`setExpr` member function
  `IloObjective` class **200**
`setLB` **82**
`setLinearCoef` **82**
setOut **295**
`setParam` **69**, **70**
`setParam` member function
  `IloCplex` class **41**
`setPriorities` **72**

`setPriority` **72**
setting
  all default parameters **41**, **99**
  callbacks to null **41**, **99**
  customized parameters **400**
  parameter specification file **400**
  parameters **41**, **98**, **400**
  see also changing
`setUB` **82**
simplex
  dual **125**
  optimizer **158**
  primal **125**
  see also dual simplex optimizer
  see also primal simplex optimizer
simplex optimizer
  parallel **379**
`SimThreads` **376**
singularity **138**
slack variable
  accessing values **43**
slack variables **133**
solution
  accessing quality information **45**
  basic infeasible primal **139**
  basis **158**
  binary files for **290**
  complementary **157**
  differences between barrier, simplex **158**
  example QP **215**
  feasible in MIPs **240**
  file format for nonbasis **292**
  incumbent **232**
  infeasible basis **172**
  midface **158**
  nonbasis **158**
  quality **163**, **164**, **169**
  supplying first integer in MIPs **240**
  text file for **292**
  verifying **169**
solution algorithm **69**
solution value
  accessing **43**
solve a single LP **69**
`solve` member function
  `IloCplex` class **39**, **42**, **44**, **45**, **46**, **53**
solve subsequent LPs or QPs in a MIP **70**
solving
  model **39**
solving the diet problem **68**
SOS
  branching priority **245**
  file format for **291**
  type 1 **37**
  type 2 **37**
  using **244**
  weights in **246**
SOS file format **245**, **291**
sparse matrix **74**
Special Ordered Set (SOS)
  see SOS
special ordered sets **61**
  single LP **69**
speed increase **375**
`square` **62**
stalling **138**
starting algorithm
  parallel processing **376**
static variables **87**
status variables, using **352**
steepest-edge pricing **132**, **254**
`string` **70**
`StringParam` **70**
strong branching **252**
`StrongThreadLim` **375**
`sum` **62**
surplus argument **308**
symbolic constants **91**, **98**

T

tail **176**
terminating
  because of singularities **138**
  because of stalling **139**
  because of time limit **147**
  MIP optimization **241**
  network optimizer iterations **179**
Threads **375**
threads **371**

clones **379**
parallel optimizers **371**
performance and **372**
thread-safe **87**
tolerance
feasibility **179**
Markowitz **138**, **139**
optimality **142**
relative optimality **252**
tolerance parameters **232**
TRE file format **242**, **254**, **292**
`TreLim` **254**
TXT file format **292**
type
conversion **82**
variable
change **61**

U

`Unbounded`
return status **67**
unbounded optimal face **159**, **171**
`Unknown`
return status **67**
upper bound values
choosing **63**
utility routine
defined **84**

V

variable
accessing dual **43**
cannot be added to a model **64**
changing type **34**, **47**
constructing an array **76**
deleting **82**
external **87**
global **87**
in expressions **34**
modeling **61**
numerical **37**
order **292**
removing from basis **46**
representing with `IloNumVar` **34**
semi-continuous **37**, **246**
static **87**
type **225**
variable selection strategy
strong branching **251**, **256**
variable type change **61**
variables
create modeling **61**
semi-continuous **61**
`VarSel` **375**
VEC file format **160**, **292**
vectors, rim **293**
violation
bound **142**
constraint **142**

W

`WorkDir` **166**, **255**
working memory **166**
`WorkMem` **166**, **242**, **254**